A Modular Verification Methodology for Caching and Lock-Based Concurrency in File Systems

Dissertation
Zur Erlangung des Doktorgrades Dr. rer. nat.
Institut für Software & Systems Engineering
Fakultät für Angewandte Informatik
Universität Augsburg

Jörg Michael Pfähler

Universität Augsburg
Fakultät für Angewandte Informatik

Reviewers:	Prof. Dr. Wolfgang Reif
	Prof. Dr. Alexander Knapp
Day of Defense:	July 9, 2018

Abstract

The Flashix project is a team effort to develop a functionally correct, crash-safe and concurrent file system for flash memory. The approach is based on encapsulated, modular components and their incremental refinement towards a realistic and executable implementation. Scala and C code is derived from the models. The file system provides strong guarantees in the presence of hardware failures and can tolerate crashes. It also performs internal operations in a concurrent thread of execution.

This thesis emerged from this large-scale verification effort and reports on the verification methodology and its practical application to the file system.

Crashes & Caching The first contribution is a modular approach for the specification and verification of crash-aware components. A component is crash-aware if it provides guarantees in the event of a power failure and subsequent recovery. A refinement theory is presented that facilitates increasing the atomicity of a component with respect to power failures incrementally. The semantics of the components capture the effect on a power failure of order-preserving write-back caches succinctly. This type of cache is common in journaling file systems. The semantics thereby ease the burden of specification significantly and the effect of a power failure propagates upwards a component hierarchy over every refinement step implicitly.

Lock-based Concurrency The second contribution is an extension of this theory to concurrent, crash-aware components. This allows clients of a component to call interface operations concurrently as well as the component itself to perform internal operations in another thread of execution in the background. Lipton reduction is used to merge several steps into one block that is executed atomically. The approach ensures that not only atomicity with respect to concurrent threads is achieved, but also atomicity with respect to the crash behavior of the component. Opportunities for Lipton reductions are applied automatically based on annotations of ownership. The ownership discipline ensures that access to a data structure is possible only if the possession of sufficient permissions is proven. Permissions to a data structure are acquired by locking the mutex or reader-writer lock that synchronizes access to the data structure and relinquished by unlocking.

Flashix File System The third contribution consists of the specification and verification of several components of the Flashix file system. The models include an erase block manager, a journal with transactions and garbage collection, a persistence layer with serialization and write-back caching for the journal and a component responsible for the atomicity of commits. The erase block manager hides the specific write characteristics of flash hardware and its error-prone nature from the rest of the file system. It performs wear-leveling concurrently in a background thread. For all components strong guarantees in the event of a power failure are proven. All proofs are mechanized in the tool KIV. Together the components comprise a coherent and working file system.

Acknowledgement

This dissertation would not exist without the continued support from my personal and professional environment.

I like to thank Prof. Dr. Wolfgang Reif for his continued support and I am grateful to him for always pushing us to take a broader and more general perspective.

Many thanks to Prof. Dr. Alexander Knapp for being a source of inspiration and knowledge and for a lot of interesting discussions about formal methods. His sense of humor is also remarkably wonderful and always a delight.

I am also very grateful to Dr. Gerhard Schellhorn for insightful discussions about semantics, refinement and concurrency. He also provided a lot of support for the use of and extensions to the KIV tool.

I had the pleasure of working with my former colleague Gidon Ernst and my current colleague Stefan Bodenmüller as a team on the Flashix project. I like to thank both of them for continually pushing the project towards completion and for having a fun time sharing the office together. Many thanks also to Bogdan Tofan for his work on the rely/guarantee calculus and linearizability in KIV.

Thanks are also due to all of the wonderful colleagues at the *Insitute for Software and Systems Engineering* for all the fun and insightful conversations about computer science, software engineering and life in general. It was and is a pleasure working with you! I am especially grateful to Johannes Leupolz for several helpful discussions and Dominik Haneberg for his support in all my teaching activities.

I am grateful to the many students who have participated in the Flashix project. I would especially like to thank Timo Hochberger for pioneering the models of an erase block manager, Stefan Lipp for extending several models to a concurrent setting, Stefan Fritsch for the initial tool support for components, Maxim Muzalyov for his early work on the C code generator, Dominik Mastaller for profiling and benchmarking the Flashix file system and suggesting several performance improvements and Philipp Franke for integrating Flashix into the Linux kernel.

To my wonderful family and friends!

Jörg Pfähler

Contents

List of Figures

1 Motivation & Overview

Summary. This chapter motivates the use of formal techniques for the verification of large, *concurrent* and *crash-aware* software systems. A system must be aware of crashes if guarantees in the event of a sudden power failure are critical. This thesis contributes formal techniques that facilitate the *incremental and modular* development of such systems by refinement of components. The approach is applied to a large-scale case study, namely the Flashix file system for flash memory. The file system provides strong guarantees in the presence of power failures and performs wear-leveling concurrently.

Contents

1.1 Motivation

The context of this thesis is the development of correct and reliable software systems with formal methods. Usually in software development, the software quality is increased by thorough and complete requirements engineering and rigorous testing. A different approach is to *prove* with mathematical methods that a software system is correct and fulfills certain reliability criteria. The idea is to develop a software product that is *correct by construction*. Using this approach, some classes of bugs are excluded by the specification mechanism. For example a strong type system guarantees that only valid instances of a class exist. In order to exclude other classes of bugs an additional proof is necessary. Problems in the software are discovered earlier in the software lifecycle and are therefore less costly [85, 95]. According to [85, Ch. 1] the cost of quality assurance exceeds the cost of programming by a factor of three for large projects (37% vs. 12% of the development effort). With the increasing size and complexity of software systems, the need for rigorous methods for the development of software therefore increases as well.

The Mars Rover Spirit [113] is an example of a system where a software bug almost led to the failure of the entire mission. The rover got stuck in a reset cycle due to a bug in its flash file system implementation. It was only recovered by clever use of some built-in debugging facilities. The incident prompted the proposal to build a verified file system for flash memory [74, 51] as a pilot project for Hoare's Grand Challenge for computing research [65].

The Mars Rover Curiosity also experienced a bug in its file system in 2013 triggering a switch to safe mode.

Other examples of critical software systems with bugs include the THERAC 25 medical electron accelerator [84], which led to several deaths, and the Ariane 5 rocket [86, 15], which crashed during launch. [72] argues that the latter incident could have been prevented with a more rigorous use of design-by-contract [93], where specifications of components are extended with preconditions, invariants, assertions and postconditions.

The motivating example for this thesis and the Flashix project as a whole is the incident with the Mars Rover Spirit. The Flashix project aims to answer the challenge [74, 51] and develop a verified flash file system. The project is a team effort, and this thesis therefore only considers and contributes a part of the file system. Several other parts of the file system are contributed by Gidon Ernst [43].

For algorithmic problems, the UBIFS file system [67] and the erase block manager UBI [55] are used as a blueprint. Together, UBIFS and UBI are a state-of-the-art file system with garbage collection and wear-leveling and are integrated into the Linux kernel. However, UBIFS and UBI are not verified. The formal verification of Flashix showed that there was a bug in the implementation of UBI, which could lead to loss of data in the event of a power failure (see Sec. 10.11).

1.2 Challenges

The challenges of developing a file system with formal methods fall into four categories. File systems must provide strong guarantees in the presence of *power failures* or other crashes. In order to increase performance file systems usually use *concurrency*. From a verification perspective, the *scale* of the project and the combination of power failures and concurrent operations requires effective methods for incremental and modular development of the file system and proper tool support. Finally, the *characteristics of flash hardware* permeate the entire design, implementation and verification of the file system for flash memory, and increase the difficulty of handling power failures correctly significantly.

1. Every file system must be *crash-aware* and provide strong guarantees in the event of a power failure. A correct recovery from after a reboot is *the* distinguishing factor of a file system from data structures in main memory. The Flashix file system employs several data structures and algorithms to guarantee that the recovery operation restores a state as close to the state prior to the power failure as possible. However, certain effects are visible to the user, because an implementation usually uses write-back caches. Such a cache delays a write access until it is necessary or enforced by the user. On the level of POSIX a flush of the caches can be requested by the user via a call to the operation *sync*. Not only user data is cached, but internal data structures as well.

2. Another aspect critical to the performance of modern file systems is concurrency. This is especially relevant for flash file systems, where garbage collection, wear-leveling and block erasure are necessary and should be performed in the background. Verification of concurrent systems is challenging because all potential interleavings of threads need to be considered and each thread must be able to cope with the interference of other threads. An additional challenge is ensuring that individual threads and the system as a whole make progress. Finally, it is critical to choose a correctness criterion that also considers crashes.

3. A file system is a large software product. Therefore the specification and verification approach needs to scale as well. The challenge is to isolate individual concerns of a file system into separate modules with well-defined interfaces and clear requirements and guarantees with respect to concurrency and power failures. Such a specification of a module is then used for the verification of its clients, facilitating their verification

significantly. The methodology must then ensure that the specification of a module may be replaced by its implementation in the context of a client, without jeopardizing the correctness of the client. Only then a modular and incremental development of a large file system is possible.

4. The fourth challenge is dealing with the specific characteristics of flash hardware. Flash memory cannot be overwritten directly. It needs to be erased first. However, the granularity of writes and erase operations differs. Individual pages may be programmed by a write operation, but only whole blocks can be erased. Therefore, updates to file system objects and internal data structures are cached and performed out-of-place. Memory is reclaimed by performing garbage collection. Flash hardware is also inherently prone to errors, such as partial writes and bit flips. Therefore, additional techniques such as wear-leveling and retries are necessary for a reliable flash file system.

1.3 Approach

The approach taken in the Flashix project is one of incremental refinement of *crash-aware, concurrent components*. We start out with a specification of the POSIX specification [3], which formalizes the guarantees of the file system, and incrementally split off concepts and provide an implementation. After several steps, the entire file system is implemented based on a specification of its assumptions about the underlying flash hardware.

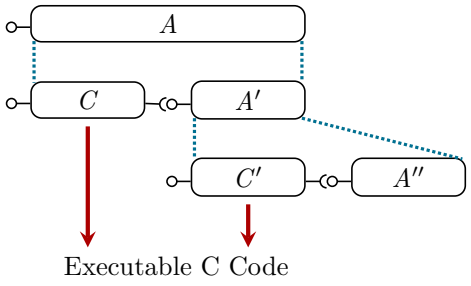

Figure 1.1: Component Hierarchies & Code Generation

Fig. 1.1 depicts the general situation, starting from an *abstract*, top-level specification component A, e.g. the POSIX specification in the case of Flashix. Each rectangle is a component, i.e., a model in a simple while- and recursion-based programming language with data types. The assumptions of used libraries or the hardware are given by the lowest specification component A'', which is a model of flash hardware in the case of Flashix. The specifications A and A'' constitute the system's boundaries. In the figure part of the implementation of the top-level specification A resides in the *concrete* component C. The component relies on the functionality of another component A', which is then implemented by the component C' based on component A''.

The specification components A, A' and A'' provide strong and easily understood guarantees with respect to concurrent threads and with respect to power failures. Correctness of an implementation C is proven locally, by considering its specification A and the specification of its subcomponent A' only. We usually say that an implementation *refines* its specification or that the implementation is a *refinement* of its specification if a proof of correctness was conducted. The blue, dotted lines in the figure denote this refinement relation.

If a system part $C' \multimap A''$ refines its specification A', then it may be substituted for its specification in any context without a change in the correctness of the context. This means that the system $C \multimap C' \multimap A''$ behaves the same way that the system $C \multimap A'$ does. If $C \multimap A'$ is a refinement of the top-level specification A, then so is $C \multimap C' \multimap A''$. We have therefore shown that the running system composed of all implementations is correct with respect to the top-level specification.

This approach ensures that we can incrementally and modularly develop a large system and at the end substitute all implementations for their specifications without affecting correctness. Such a verification methodology scales, since proofs only have to consider a small, well-defined part of the entire, complex implementation.

The approach also facilitates code generation. From the implementation models C and C' in the figure code in the C and Scala programming language is generated (red arrows) that provides the interface of A and uses the interface A''.

1.4 Contributions of this Thesis

This thesis contributes a specification and verification methodology for large, crash-aware and concurrent systems, a large part of the specification and verification of the Flashix file system and the actual, running file system. In the following the contributions in each part are highlighted.

Specification Methodology for Crash-Aware, Concurrent Components From a theoretical perspective, this thesis contributes a novel semantics for crash-aware, concurrent components. Components are modular units of software with a hidden state and a clearly defined interface. They are used as a mechanism for the specification as well as the implementation of system parts. The semantics of components is compositional and facilitates the modular and incremental development of larger systems.

A power failure is visible in the semantics as an additional transition from any intermediate state of an operation. There are two specification mechanisms to describe the effect of this transition. First, there is the *state-based* approach, where all variables in volatile memory are set to arbitrary values, and (an abstraction of) the persistent state is modified according to the crash predicate. Afterwards, the recover operation of the component is run. Only after recovery, the component can service requests by its client components again. This state-based specification mechanism corresponds naturally to the effect of a power failure. From a specification perspective, it is used for minor, local effects of a power failure that are visible to client components. In the Flashix case study the state-base view is mainly used to propagate the effects that write-back caching of several internal data structures has on part of the component hierarchy.

The second approach is *operations-based*. The effects of several operations before a power failure are *retracted* and some operations may be completed with a different output. In order to limit the number of operations that may be retracted, *synchronized states* are used, i.e., an operation that yields a synchronized state is never retracted as an effect of a power failure. This specification mechanism is useful for effects that are visible over a hierarchy of several components. The advantage is twofold. First, it is much easier to understand from the perspective of the user. Second, the effect propagates upwards a refinement hierarchy *implicitly* with only trivial additional proof obligations. In this thesis this view is used to propagate the effect of write-back caching of user data.

The implicit, operations-based approach is completely novel. The crash-aware, concurrent components contributed by this thesis support the state-based as well as the operations-based specification mechanism.

Verification Methodology for Crash-Aware Components As a criterion for correctness of an implementation *crash linearizability* with respect to a sequential specification component is introduced in this thesis. Incremental refinement towards a sequential specification is used as a verification methodology. Refinement preserves termination and crash linearizability. In this thesis it is proven that refinement allows for the substitution of an implementation component for its specification in a context while preserving the correctness of the context.

For sequential, crash-aware components three types of refinement are used: *atomicity refinement*, *data refinement* and *crash refinement*. Atomicity refinement facilitates composing large atomic blocks that guarantee atomicity with respect to power failures. This reduces the number of states power failures have to be considered in. This thesis contributes a novel calculus to determine whether a sequential program may be replaced by an atomic block.

This thesis also contributes a variant of data refinement that propagates the operations-based view on a power failure implicitly. The notion of crash refinement is also completely novel and facilitates switching from the state-based to the operations-based view of a power failure.

For concurrent, crash-aware components, atomicity refinement is extended by and integrated with Lipton reductions [87]. A Lipton reduction facilitates substituting a critical region protected by a mutex or reader/writer lock with an atomic block that guarantees atomicity with respect to concurrent threads. This thesis contributes a variant of Lipton reductions that take atomicity with respect to power failures into account as well.

Verification of Flash File Systems This thesis contributes several components critical to a state-of-the-art flash file system. For each component correctness is proven using the verification methodology of incremental refinement. The aspect that permeates all models is atomicity of some concepts with respect to power failures and write-back caching of user data or internal data structures.

This thesis contributes the following components, concepts, and their verification:

- A model of flash hardware compliant with the industry standard ONFI [4] (Open NAND Flash Interface), which formalizes the assumptions about the hardware.
- An erase block manager that performs wear-leveling and asynchronous erasure of blocks concurrently in the background. The component increases the reliability of the file system by managing bad blocks transparently and moving contents when errors are detected. The mapping from logical to physical blocks maintained by the erase block manager is write-back cached.
- A write-buffer that caches file system objects until a full flash page can be written.
- Serialization and deserialization of individual file system objects and detection of partially written file system objects.
- Journaling with atomic transactions, each of which consists of modifications to several file system objects, and garbage collection of obsolete versions of file system objects.
- Atomicity of the file system commit, where several internal data structures are persisted. Updates to the internal data structures are also write-back cached up to a commit.

The individual components are described in Ch. 6 in more detail.

Flashix File System The final contribution of this thesis and the thesis of the author's colleague Gidon Ernst [43] is the working C code of the Flashix file system. The code is generated from the implementation components and integrated into the Linux kernel. Ch. 2 provides more detail on the code and its integration.

Publications The work in this thesis has led to and contributed to the following publications.

1. J. Pfähler, G. Ernst, G. Schellhorn, D. Haneberg, and W. Reif. Formal specification of an Erase Block Management Layer for Flash Memory. In *Haifa Verification Conference (HVC)*, volume 8244 of *LNCS*, pages 214–229. Springer, 2013
2. G. Ernst, G. Schellhorn, D. Haneberg, J. Pfähler, and W. Reif. A Formal Model of a Virtual Filesystem Switch. In *Proc. of Software and Systems Modeling (SSV)*, EPTCS, pages 33–45, 2012
3. G. Ernst, G. Schellhorn, D. Haneberg, J. Pfähler, and W. Reif. Verification of a Virtual Filesystem Switch. In *Proc. of Verified Software: Theories, Tools, Experiments (VSTTE)*, volume 8164 of *LNCS*, pages 242–261. Springer, 2013

4. G. Schellhorn, G. Ernst, J. Pfähler, D. Haneberg, and W. Reif. Development of a Verified Flash File System. In *Proc. of Alloy, ASM, B, TLA, VDM, and Z (ABZ)*, volume 8477 of *LNCS*, pages 9–24. Springer, 2014. Invited Paper
5. G. Ernst, J. Pfähler, G. Schellhorn, and W. Reif. Modular Refinement for Submachines of ASMs. In *Proc. of Alloy, ASM, B, TLA, VDM, and Z (ABZ)*, pages 188–203. Springer, 2014
6. G. Ernst, J. Pfähler, G. Schellhorn, D. Haneberg, and W. Reif. KIV - Overview and VerifyThis Competition. *Software Tools for Techn. Transfer*, 17(6):677–694, 2015
7. G. Schellhorn, B. Tofan, G. Ernst, J. Pfähler, and W. Reif. RGITL: A temporal logic framework for compositional reasoning about interleaved programs. *Annals of Mathematics and Artificial Intelligence*, 71(1):131–174, 2014
8. G. Ernst, J. Pfähler, G. Schellhorn, and W. Reif. Inside a Verified Flash File System: Transactions & Garbage Collection. In *Proc. of Verified Software: Theories, Tools, Experiments (VSTTE)*, volume 9593 of *LNCS*, pages 73–93. Springer, 2015
9. G. Ernst, J. Pfähler, G. Schellhorn, and W. Reif. Modular, Crash-Safe Refinement for ASMs with Submachines. *Science of Computer Programming (SCP)*, 2016
10. J. Pfähler, G. Ernst, S. Bodenmüller, G. Schellhorn, and W. Reif. Modular Verification of Order-Preserving Write-Back Caches. In *IFM: 13th International Conference, 2017, Proceedings*, pages 375–390. Springer, 2017

1.5 Outline

Fig. 1.2 shows an overview over the chapters of this thesis and their interdependencies. In the following the above contributions are connected to the corresponding chapters.

Ch. 2 provides an overview over the case study, namely the Flashix file system, and Ch. 3 provides the theoretical background that forms the basis of the verification methodology.

Ch. 4 provides details on the syntax and semantics of crash-aware, concurrent components. Furthermore, a compositionality result is proven, i.e., it is proven that an implementation can be substituted for its specification in a context.

In Ch. 5, three types of refinement for sequential, crash-aware components are discussed. We usually first perform an *atomicity refinement* to increase the atomicity of a component with respect to power failures. Afterwards, a *data refinement* is used to change the representation of data structures. Finally, a *crash refinement* facilitates switching from the state-based to the operations-based view for power failures.

Ch. 6 gives an overview over the components of the Flashix file system and how the verification methodology for sequential, crash-aware components is applied to the case study. The focus of the chapter is that crash-safety of a file system is a cross-cutting concern, which permeates the entire file system and each level of abstraction has to deal with and recover from power failures in the middle of an operation.

Chapters 7 and 8 present the system boundaries of the Flashix file system formally as components. The component *POSIX* formalizes the guarantees of the file system, whereas the component *Flash* encodes the assumptions about the flash hardware and its driver.

Ch. 9 briefly explains how serialization and deserialization procedures are integrated into the formal development and the process of code generation.

Ch. 10 shows an abstract specification and implementation of an erase block manager. An erase block manager increases the reliability of the flash hardware by retrying operations and moving data if errors are detected. It also implements wear-leveling, which ensures that blocks are erased evenly and further increases the reliability and lifespan of flash hardware. For performance reasons, the erase block manager erases blocks asynchronously. The chapter provides some detail on the invariants of the erase block manager and its verification.

In Chapter 11, the core concepts of a flash file system are presented, namely journaling with transactions consisting of modifications to multiple file system objects and garbage collection of obsolete versions of file system objects. The integration with an efficient index

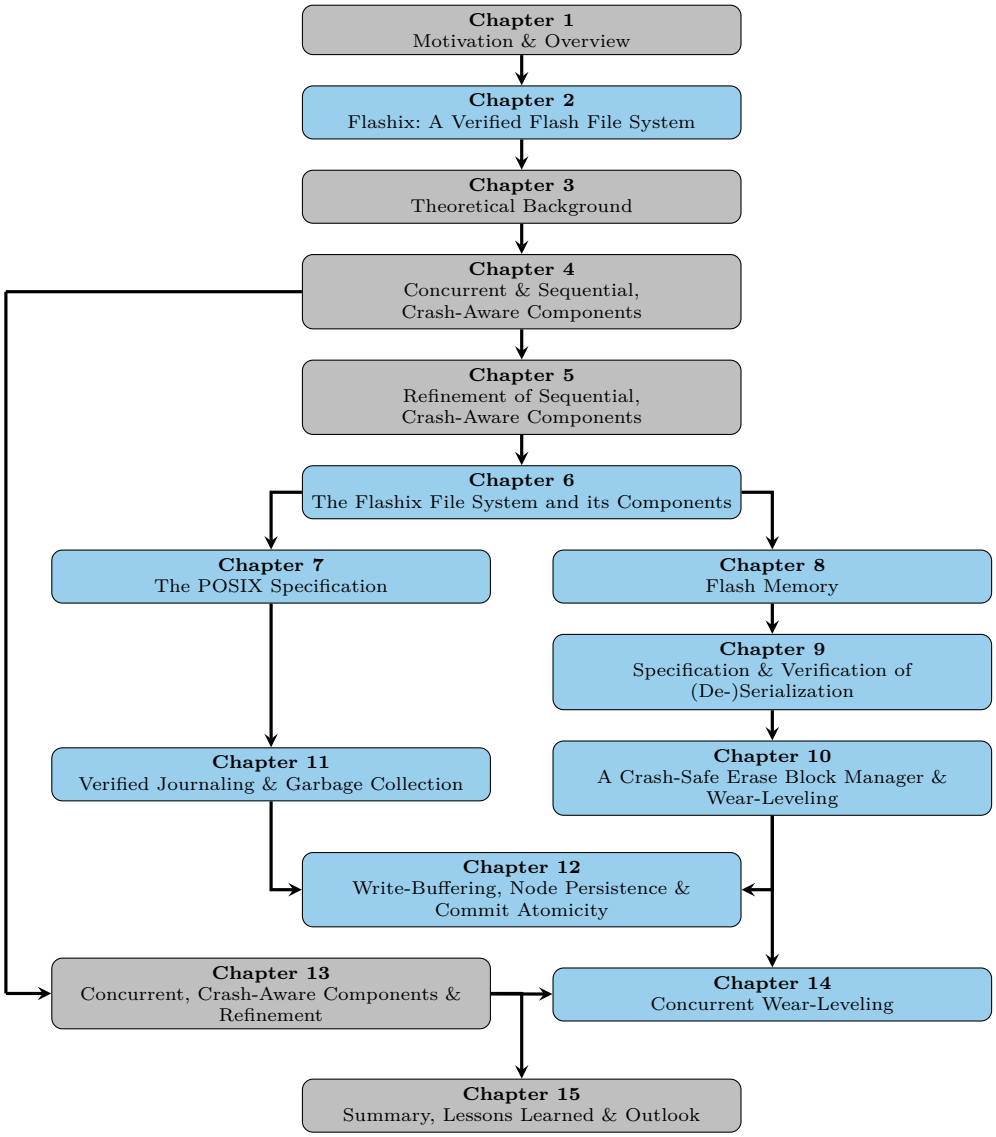

Figure 1.2: Overview over the Structure of the Chapters and their Dependencies: The chapters describing parts of the Flashix file system are depicted in blue.

to locate the current version of a file system object is discussed. It is proven that transactions are atomic, and that garbage collection is correct and picks a suitable block.

Individual file system objects are serialized to bytes in Ch. 12. It is proven that the byte representation ensures that partially written nodes are detected and discarded. This guarantees that individual file system objects are written atomically. Writes of the file system objects are cached by a write-buffer to increase performance and to deal with the write characteristics of flash hardware. For this component a crash refinement is used to switch to the operations-based view on a power failure. This facilitates propagating the effect of this write-back cache upwards the entire refinement hierarchy. The chapter also proves the correctness and atomicity of the file system commit, when several internal data structures such as the index are written to flash memory.

Ch. 13 extends the specification and verification methodology introduced in Ch. 5 to concurrent components. It is proven that with a variant of Lipton reductions it is possible to

increase the atomicity of a component with respect to concurrent threads as well as power failures.

In Ch. 14 the verification methodology is applied to the erase block manager, which shows that wear-leveling and asynchronous erasure of blocks can be performed concurrently by a separate thread.

Ch. 15 summarizes the results and contributions of this thesis, discusses some lessons learned and gives an outlook for the Flashix project.

2 Flashix: A Verified Flash File System

Summary. This chapter gives an overview over the Flashix file system, which is a verified file system for flash memory. The file system is functionally correct and crash-safe with respect to the POSIX specification. Wear-leveling and asynchronous erasure of blocks are performed concurrently in the background. Crash-Safety means that the file system deals with power cuts in a guaranteed and predictable way. Although the development is based on models, final C code is generated and is integrated as a Linux kernel module.

Publications. This Chapter is based on the overview over the Flashix project published in [118].

Contents

2.1 Overview over the Development

The Flashix file system is a file system for flash memory. It provides the POSIX interface [3] to client programs and uses the MTD (= Memory Technology Devices) interface to interact with flash hardware as shown in Fig. 2.1. MTD is the standard interface for flash devices in Linux-based operating systems. MTD is implemented by drivers for *raw* flash devices, usually found in embedded systems.

Solid State Drives (= SSDs) are outside the scope of the project, although SSDs are now more common than raw flash devices in consumer electronics. A Flash Translation Layer (= FTL) that simulates the interface of a normal magnetic disk is usually used for SSDs in conjunction with a traditional file system. As observed by Yang et al. [139] the direct approach of a flash file system provides performance benefits. In mission-critical system, such as the Mars Rover Spirit, usually raw flash is used, because SSDs need a lot of additional

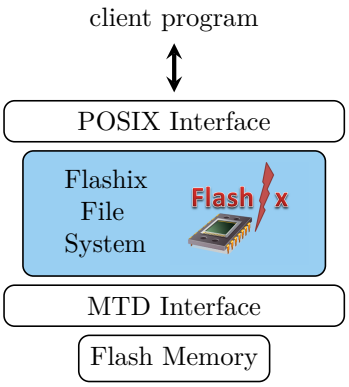

Figure 2.1: The Flashix Project: A Verified File System for Flash Memory

electronics and logic on-chip.

Flashix is developed with the interactive verifier KIV[1] [44] by incremental refinement of the POSIX specification. The correctness of the file system is ensured by construction. For the specification and implementation models *crash-aware, concurrent components* are used. Ch. 4 provides more detail on the formalism. From each of the implementation models executable C and Scala code is generated and integrated into Linux-based operating systems. Flashix is therefore usable as a normal file system on Linux.

The formal development of Flashix encompasses a lot of concepts that are independent of the concrete file system and many aspects of general interest to any flash file system. Fig. 2.2 depicts the concepts and aspects that are considered. The rectangles do not immediately correspond to a crash-aware component. The hierarchy of components is shown in Ch. 6 (see Fig. 6.1 on page 66) after the formalism and approach is explained in more detail.

System Boundaries The POSIX specification at the top of the figure specifies the syntactic interface, i.e., the operations with input and output parameters, provided by the Flashix file system. A syntactic interface, however, is not enough. The POSIX specification also formalizes the semantics of each of the operations, i.e., the allowed inputs and more importantly all allowed outputs. Essentially, the specification is a formalization of the guarantees given by the Flashix file system, or any other correct file system. The most important part of the specification is the effect of power failures on the state of the file system and on the visible behavior after the recovery from the crash. The crucial correctness criteria for a file system, and the distinguishing factor from any implementation that only uses main memory to store the directory structure and file contents, is that it implements this crash behavior correctly. At the lowest level, the flash driver specification MTD formally defines the syntactic interface expected from a flash driver as well as all assumptions about the behavior of flash hardware the Flashix file system and its correctness relies on. The formal models of the system boundaries are presented in Ch. 7 and Ch. 8.

Guarantees Informally, the Flashix file system guarantees that operations terminate and implement the POSIX standard faithfully. With respect to power failures, the file system guarantees that operations are executed atomically. Since write-back caches are employed, not all of the completed operations already had a persistent effect. As an effect of a power failure therefore several of the last operations are retracted and one may be re-executed completely. This yields an alternative execution of the file system up to the power failure that is consistent with the observable behavior after the power failure, but has executed several operations less and one operation differently. The file system also guarantees that a successful call to the POSIX operation `sync` is never retracted or re-executed, because it yields a synchronized state. The file system may only be called from one thread currently. However, internally Flashix performs wear-leveling and asynchronous erases in a background thread concurrently, without disturbing other file system operations.

Assumptions The assumptions about flash hardware are essentially that errors are detected and reported faithfully by the flash driver and the underlying hardware. This means that an error is returned when reading or writing fails, and no error is returned if it does not fail. The hardware may always fail to perform an operation without modifying the state. For writes Flashix assumes that only page-aligned partial writes are possible, to be able to reliably detect partially written headers and file system objects. With respect to power failures, the assumption is that no additional effect on the state is visible, i.e., that the raw flash device does not employ any additional caches.

Concepts & Layers The Virtual File System Switch provides all file system independent functionality, such as path lookup, handling of opened files with the current offset for reads

[1] http://www.isse.uni-augsburg.de/software/kiv/

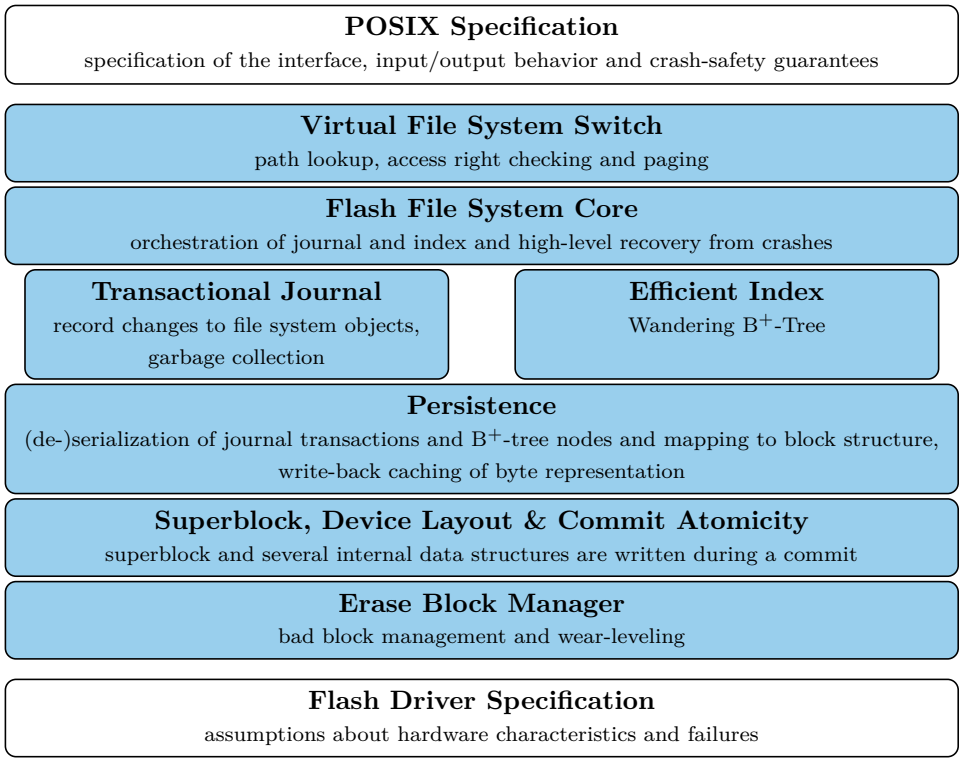

Figure 2.2: High-level Overview over the Flashix File System: At the highest level the *POSIX Specification* formalizes the guarantees given by Flashix. At the lowest level the *Flash Driver Specification* precisely expresses the assumptions about the hardware.

and writes, and segmentation of files into individual pages. Path lookup traverses the directory structure of the file system until the requested file or directory is found. The VFS communicates with the Flash File System Core and requests information about individual file system objects. A file system object is either an inode, a directory entry or a page of a file. An inode holds meta data, such as access times or permissions, about one specific file or directory. A directory entry forms the edge of the directory tree. It stores the name of the directory entry and connects the inode of the parent directory with the child inode.

The Flash File System Core implements a journaling and log-structured file system [116] in order to cope with power failures and with the write characteristics of flash hardware. Updates of file system objects are written out-of-place, i.e., the old version of a file system object is not overwritten, but a new version is written to a sequential journal. The journal essentially stores all updates to file system objects since the last commit. The current version of a file system object is located with an index, implemented as a wandering B^+-tree. The operations of a file system usually do not only modify a single file system object. The creation of a file for example updates the parent directory inode, and adds a new directory entry and a new file inode. Atomicity of these three updates with respect to crashes is guaranteed by employing a transactional journal.

The transactional journal ensures that transactions are indeed atomic with respect to power failures, i.e., partially written transactions need to be detected and discarded. The layer also performs garbage collection of blocks with a lot of obsolete versions of file system objects.

The persistence layer serializes individual file system objects and ensures that partially written file system objects are detected. It employs a page-sized write-back cache, the write-buffer, to write serialized file system objects to a flash block. This buffer is necessary, because

flash hardware allows page-aligned writes within a block only. The buffer increases memory utilization and performance by around a factor of two. However, the effect of losing a part of the data is visible across the entire file system. At each level above the write-buffer, the effect of a power failure corresponds to a retraction of several operations and a re-execution of one operation up to the level of the POSIX specification. Such a write-buffer is used in most journaling file systems and is therefore not unique to flash memory or Flashix. Its effect is the main motivation for the component semantics of Ch. 4, which implicitly propagate the effect of a power failure of an order-preserving, write-back cache upwards an entire hierarchy of components. An order-preserving cache writes data to persistent storage in the order of the client's requests.

Conceptually below the write-buffer component, the first few blocks of the flash device are reserved for the superblock and two versions of the commit data structures. The commit data structures are written during a file system commit and consist of an on-flash version of the index and several other internal data structures. Atomicity of the commit is guaranteed by provisioning space for two versions of the data structures. The data structures are written out-of-place and if successful, then a final, atomic write to the superblock switches from the old version of the commit data structures to the new version. The commit data structures are also write-back cached, since all updates are first performed in main memory and only during a commit a new version is persisted.

The lowest layer is the erase block manager. It manages bad blocks, erases blocks asynchronously in the background and performs wear-leveling. In case of errors during writing or reading, the layer tries to move the contents of an erase block to a new location. The erase block manager thereby increases the reliability of flash hardware significantly and forms the basis of a robust flash file system. In order to transparently move blocks and perform wear-leveling, an abstraction of logical erase blocks is presented to clients. Internally, a mapping from logical to physical erase blocks is maintained. This mapping is also write-back cached, i.e., updates to the version of the mapping in main memory are not immediately persisted. The in-RAM and on-flash version of the mapping correspond only once all blocks where an asynchronous erasure was requested are actually erased.

The Flashix project is a team effort and this thesis does not contribute the implementation and verification of all the concepts and aspects shown in Fig. 2.2. Specifically, the POSIX specification, the implementation of VFS, the wandering B^+-tree as an index and the Flash File System Core are contributed by the author's colleagues Gidon Ernst [43] and Andreas Schierl [121]. Ch. 6 details the specific components that form the contribution of this thesis with respect to the Flashix file system.

2.2 Code Generation & Linux Integration

The components are modeled in KIV as abstract programs over algebraic data types and are not immediately executable on normal hardware. Code is generated from the models and integrated into Linux-based operating systems. For initial testing and debugging Scala [103] code is generated, which provides built-in support for algebraic data types and runs on the Java Virtual Machine (= JVM). The generated code is then integrated via the *File System in Userspace* (= FUSE, [125]) Linux kernel module, which facilitates developing file systems as an application outside of the Linux kernel. FUSE is usable from Scala with the Java bindings called FUSE-j [83]. The left-hand side of Fig. 2.3 shows the FUSE integration. The generated code of Flashix itself is around 7.3kLoC, and the integration with FUSE accounts for an additional 0.7kLoC.

However, Scala is obviously a bit slower and not suited for small embedded devices, which is usually targeted by a file system for raw flash. Therefore, C code is generated for the Flashix file system, too. It is integrated via FUSE similarly to the Scala integration as well as directly into Linux as a kernel module. The right-hand side of Fig. 2.3 shows the direct

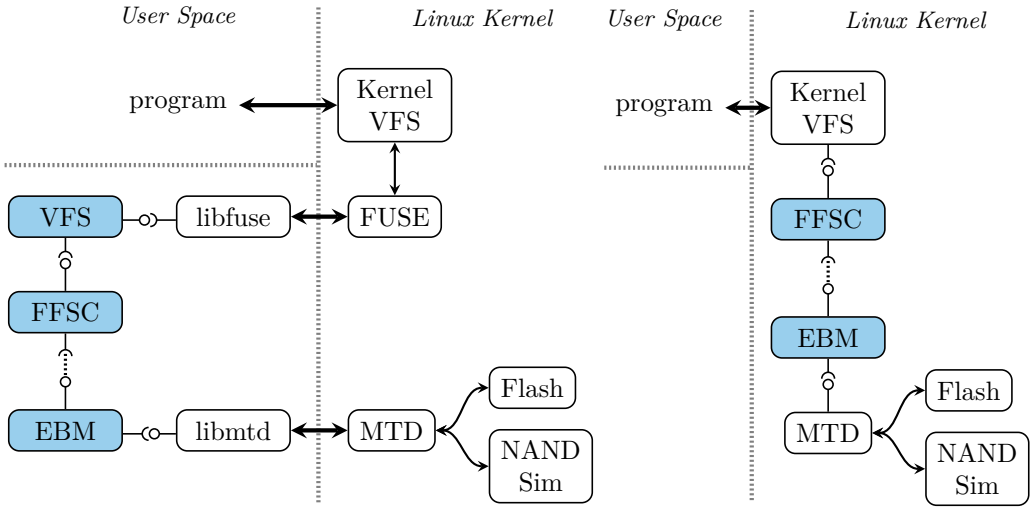

Figure 2.3: FUSE-based (left) and Direct Integration (right) of Flashix into Linux-based Operating Systems: The dotted line represents address space boundaries. The FUSE-based implementation is path-based and therefore the Virtual File System Switch model (= VFS) is used. The direct integration is inode-based and uses the Flash File System Core (= FFSC).

integration. In the direct integration one of the components of Flashix, namely the Virtual File System Switch, is replaced by the corresponding implementation of the Linux kernel. The C code for the file system itself has around 13.4kLoC, the FUSE-based integration and kernel integration add an additional 1.9kLoC and 2.5kLoC, respectively.

According to the developers of FUSE, the integration of a file system via FUSE is significantly slower compared to a direct integration as e.g. observed by [112] and our own measurements. Note that this depends on the workload and on whether hardware latency is the problem or memory bandwidth, e.g. if a flash device is only simulated in memory then hardware latency obviously plays no role and FUSE becomes significantly slower due to a lot more context switches and memory copying operations.

As already shown in Ernst [43], the Flashix file system is comparable to UBIFS with respect to performance for microbenchmarks. However, in benchmark tools such as iozone[2] [100] Flashix is still an order of magnitude slower than UBIFS. The reason is that it does not yet use the write-back cache of the VFS of Linux, or a comparable write-back cache. Initial experiments with a modified kernel integration, which additionally uses the VFS cache but is not yet complete, suggest that the performance is then en par with UBIFS. These write-back caches are, however, not in the scope of this thesis, but are currently being modeled and proven correct. A detailed performance analysis of Flashix therefore remains future work.

2.3 Related Work

An empiric study of Linux file system bugs of Lu et al. [88] shows that most errors in file systems (around 90%) are either categorized as semantic, concurrency or memory management. In a correctness by construction approach most of these errors are impossible. The fourth kind of error, namely the return of wrong error codes, however, is still possible, because an abstract specification usually allows for all low-level errors, such as `ENOSPC`, to be returned.

Log-structured file systems first appeared in Rosenblum et al. [116]. A log-structure has since been integrated in several file systems. Most notably the flash file systems JFFS [135], JFFS2 and UBIFS [67] use a log-structure to deal with the write characteristics of flash hardware and as a measure for the recovery from power failures. The more general form of a

[2] http://www.iozone.org/

journaling file system is quite prevalent in file system for hard disk. For example NTFS and Ext3 [130] both employ the technique.

Prabhakaran et al. [111] have tested several traditional file systems with respect to failures by injecting disk failures and found that most file systems handle recovery from failure inadequately and inconsistently. With their gained experience, they developed the file system ixt3 based on ext3 with improved robustness. Ridge et al. [115] developed the model-based testing framework SibylFS for POSIX-compliant file systems.

With model checking Yang et al. [140] could find 32 bugs in several Linux file system in total, all of which could lead to unrecoverable loss of data. Model checking is also applied to the Virtual File System Switch of Linux in [54] by manually extracting a model from the C code and applying SPIN [66]. The approach ensures that the model maintains certain integrity properties with respect to its internal data structures and that the model is dead-lock free. In [97] static analysis is applied to the compiled code of the Linux VFS, which proves the absence of memory management related bugs, such as dereferencing of invalid pointers, for several of its operations. Koskinen et al [80] use model checking to ensure that a system is crash recoverable. They apply the tool to large database system such as PostgreSQL and inject several faults.

Marić and Sprenger [90] model crashes as exceptions, where the exception handler performs a recovery of the system. The system they consider is a sequential, transactional memory manager. The interrupt operator in CSP [92] could be used for this purpose. In [92] algebraic laws for the interrupt operator are proven correct.

In [101] a variant of separation logic [114] is introduced that facilitates reasoning about volatile and durable memory by using a separate formula for each. In [124] a logic for file systems is introduced. The logic also separates formulas that refer to the volatile state from formulas about the durable state. This allows them to express data integrity at a very high-level of abstraction.

In [123] the Yggdrasil tool is introduced, which automatically checks whether a file system is a crash refinement of their file system specification. The approach employs SMT solvers as a reasoning engine. An asynchronous disk model, which potentially re-order writes, is used. The top-level specification does not allow for write-back caching across several operations. With the tool they verify their file system Yxv6.

Bornhold et al. [18] provides a model for POSIX that includes a specification of power failures. Several correctness criteria, termed crash-consistency models, are introduced. They developed the Ferrite tool, which facilitates checking Linux file system against crash-consistency models. [110] presents an overview over the problems associated with crashes and re-ordering of modern file systems from the perspective of application writers.

Several other projects produced a verified file system, too.

FSCQ [27, 28, 26] is a file system for hard disks and is proven functionally correct and crash-safe in the Coq proof assistant [17]. Its design follows the xv6 file system [35]. A novel form of Hoare logic, termed Crash Hoare logic, is used and augments standard Hoare triples with crash conditions. Via Coq's code extraction Haskell code is automatically derived from the specifications.

Amani et al. [77, 11, 9] develop the Bilbyfs file system. The file system is for flash memory, and uses a design similar to Flashix. Bilbyfs also employs a write-buffer to deal with the write characteristics of flash hardware. The focus of the research is the correctness of the C code generation. Their approach to specification facilitates the automatic generation of C code and the generation of an accompanying proof of correctness with the tool Cogent [10, 102]. The specification and verification is conducted in the interactive proof assistant Isabelle/HOL [99]. Crash-safety is outside the scope of the verification.

Damchoom et al. [39, 37, 36] use Event-B [7] to develop a verified flash file system with incremental refinement and machine decomposition. Concurrency of read and write operations on the level of AFS is covered by the verification. Their idea is to split a read and write operation into several concurrent read and write operations of smaller sizes. The

synchronization between threads is implicitly performed by the semantics of Event-B models. Java code is derived manually from the Event-B models.

There are several other large-scale verification efforts. Leroy [82] proved that the C compiler CompCert, which translates a subset of the C programming language to PowerPC, is correct. The CakeML project[3] [81, 126] provides a verified compiler for a subset of StandardML to several processor instruction sets. NICTA has produced a verified operating system kernel [78]. The mondex case study is a medium-sized verification effort conducted by several groups [134, 119, 20].

[3] https://cakeml.org/

3 Theoretical Background

Summary. This Chapter summarizes the theoretical basis for this thesis. The syntax and semantics of programs is presented and the utilized logics and tooling are introduced. The logics are supported by the interactive proof assistant KIV.

Publications. The description of the KIV verification system and its support for algebraic specifications, data types and sequential programs is based on [44]. The Chapter extends the logic RGITL [120] with atomic sections and assertions.

Contents

3.1 Interval Temporal Logic

The logic RGITL [120, 127] integrates rely/guarantee reasoning and interval temporal logic [25]. It is implemented in the interactive proof assistant KIV [44] and has been used successfully to verify lock-free and linearizable algorithms and data structures.

The logic is based on intervals with system and environment steps with static variables x and flexible variables X. Static variables are written lowercase and flexible variables start with an uppercase letter. Static variables are used to capture the value of a flexible variable at some point in time. An *interval* $I = (I(0), I_b(0), I(0)', I(1), \ldots)$ is a finite or infinite sequence of alternating *system* and *environment* steps. A system transition starts in state $I(i)$ and yields state $I(i)'$. It is *blocked* if $I_b(i)$ is $\mathtt{tt}$. The transitions from $I(i)'$ to $I(i+1)$ are performed by the environment. The last transition is always an environment transition. Each state of the interval assigns a value to each static and flexible variable. For static variables the value of a variable is the same in every state of the interval. The length of an interval I is denoted by $\#\, I$ and defined as the number of system transitions if the number of states is finite. Therefore an interval with n system steps has $2 \cdot n + 1$ states. The first state of an interval is often written as $I.\mathit{first}$. If the interval I is finite then $I.\mathit{last}$ denotes its last state, i.e., the state $I(\#\, I)$.

The concatenation of two intervals I_0 and I_1 is written $I_0 \fatsemi I_1$. For finite I_0, concatenation is possible only if $I_1.\mathit{first} = I_0.\mathit{last}$ holds. If I_0 is infinite, then $I_0 \fatsemi I_1$ is equal to I_0. The suffix after n system and environment steps of an interval I is written $I_{[n..]}$. Similarly, the prefix up to and including the n-th and environment step is denoted by $I|_n$. The infix $I_{[n..m]}$ is defined

as $(I|_m)_{[n..]}$ for $n \leq m$.

The expressions and formulas of RGITL include the standard constructs of predicate logic and interval temporal logic, such as $\Box\varphi$ and $\diamond\varphi$. Flexible variables can be unprimed, primed and double primed, i.e., X, X' and X'' are valid formulas. The expressions X and X' refer to the value of the variable after resp. after the system step, and X'' evaluates to the value after the environment transition. For example the formula $\Box\left(X' = X + 1 \wedge X'' = X\right)$ states that the system steps always increase the value of X by one. The environment steps leave the value of X unmodified.

Substitution of the variable x by the term t in a formula φ is written φ_x^t. Substitution of a static variable x with a value v in an entire interval I is denoted by $I(x \mapsto v)$. The value of a flexible variable X in some state $I(i)$ and $I'(i)$ of the interval is written $I(i)(X)$ and $I'(i)(X)$, respectively. Evaluation of a predicate logic expression e in a state $I(i)$ of the interval is written $[\![e]\!](I(i))$. Evaluation of a temporal logic formula φ over an interval I is written $I \models \varphi$. For predicate logic formulas φ evaluation in a state $I(i)$ is denoted by $I(i) \models \varphi$.

Definition 3.1 (Empty Environment)**.** An interval has an *empty environment* if all environment steps leave all variables unmodified, i.e., if $I'(n) = I(n+1)$ holds for all $n < \#\, I$.

Tuples are written with angle brackets $\langle\_\rangle$ in the following. The empty tuple is $\langle\rangle$. A variable $\underline{x}$ of tuple type is indicated by an underline and its individual components are referred to as x_i.

3.2 Program Syntax & Semantics

This section first defines the syntax of programs. The semantics of a program p is given by the set of intervals of the previous section where the system steps satisfy the changes to variables made by p.

Definition 3.2 (Program Syntax)**.** A program follows the syntax given by the following grammar.

$p, q :=$	$\underline{X} := \underline{t}$	simultaneous assignment
	$p; q$	sequential composition
	choose* $\underline{X}$ **with** φ **in** p **ifnone** q	nondeterministic choice
	if* φ **then** p **else** q	if-then-else
	while φ **do** $\{\, p \,\}$	while
	$\texttt{Proc}(\underline{t}; \underline{X})$	procedure call
	atomic φ $\{\, p \,\}$	atomic block
	assert φ	assertion
	$\Big\|\Big\|_{Tid \in \{1,\ldots,n\}} p$	weak-fair interleaving
	p^*	finite or infinite iteration

All expression contained in programs are predicate logic expressions that do not contain primed or double-primed variables or temporal logic constructs. The condition $\varphi \equiv \mathit{true}$ may be omitted from the nondeterministic choice and atomic program constructs.

The usual program constructs of an imperative programming language are part of the syntax as defined by Def. 3.2 with a few alterations and additions.

Assignments might assign several variables simultaneously.

A **choose***-program introduces several local variables $\underline{X}$, which satisfy the condition φ, and executes p. If no such variables exist then q is executed. Note that the asterisk (*) in **choose*** denotes that choosing the variables requires no extra step before p or q is executed. This is useful for specification purposes and is e.g. required for the semantics of components as discussed in Ch. 4 and 13. We will define a corresponding version **choose** as an abbreviation.

Atomic blocks **atomic** $\varphi\ \{\, p \,\}$ serve two purposes. Firstly, the program blocks until the condition φ is satisfied. We call φ the guard of the atomic block. Afterwards, the program p is executed in a single, indivisible step. This facilitates modeling mutexes for example. Locking a mutex blocks until the mutex is free and immediately afterwards locks the mutex. Atomic blocks with a waiting condition are also composable into larger atomic blocks in order to move from a very fine-grained verification to a sequential verification if only a single atomic block is left. This technique is discussed in Ch. 5 where atomicity with respect to power failures is proven and in Ch. 13 where atomicity in the context of concurrent threads is examined.

A call to the procedure `Proc` with input parameters $\underline{t}$ and reference parameters $\underline{X}$ is written $\texttt{Proc}(\underline{t}; \underline{X})$. The value parameters may be arbitrary expressions, but the reference parameters must be variables. Modifications of input variables by the procedure are invisible to the caller, only changes to reference variables can be observed. The types of the actual parameters of the call must match the types of the formal parameters in the declaration of the procedure. Every procedure `Proc` has a declaration of the form $\texttt{Proc}(\underline{Y}; \underline{Z})\ \{\, p \,\}$ with disjoint lists of variables $\underline{Y}$ and $\underline{Z}$, where p is a program with $\mathit{free}(p) \subseteq \underline{Y} \cup \underline{Z}$. The free variables of a program p, written $\mathit{free}(p)$, are all variables of p that are not bound by a **choose***-statement. Thus, there is no global state that is implicitly passed to a procedure and (potential) effects of a procedure call are syntactically visible at the call site.

An assertion **assert** φ ensures that φ holds before a next statement p is executed. If φ is not satisfied then the program does not terminate and stutters. The aim of assertions is to ensure that they are guaranteed to hold in every execution. Then φ may be used as an assumption for further steps in the verification. This is employed in Ch. 4 and 13 to prove some property about the program p, where p is a small part of an procedure of an entire component, using φ as an assumption and then infer that p may be replaced by **atomic** $\{\, p \,\}$ in the context of the entire component. We will ensure that assertions hold by verifying termination in Ch. 4.

The program $\|_{\mathit{Tid} \in \{1,\ldots,n\}}\ p$ of n threads for a fixed number n, interleaves the execution of several programs p with free, unassigned variable Tid. The interleaving is weakly fair, i.e., it is assumed that the scheduler will run a thread that does not block persistently, eventually. The precise definition of interleaving of intervals is given in [120].[1]

Iteration p^* finitely or infinitely often executes p in sequence. The constructs is only used to specify the semantics of a component precisely, but are not used in *regular* programs.

Notation (Program Substitution). The substitution of variables $\underline{X}$ by variables $\underline{Y}$ is denoted by $p\{\underline{X} \mapsto \underline{Y}\}$. Standard rules for renaming of local variables apply, i.e., the capture of any of the variables $\underline{Y}$ by **choose***-statements needs to be avoided. Similarly, $p\{q \mapsto q'\}$ denotes the program p where all subprograms q are syntactically substituted by q'.

Definition 3.3 (Extended Program Syntax)**.** In addition to the program constructs of

[1] Technically, [120] gives semantics for the interleaving of two intervals. However, the procedure spawn_n can be used to construct a system of n interleaved threads with corresponding thread identifiers.

Def. 3.2 the left-hand side of

$$\begin{aligned}
\textbf{skip} &\equiv \langle\rangle \coloneqq \langle\rangle \\
\textbf{choose } \underline{X} \textbf{ with } \varphi \textbf{ in } p \textbf{ ifnone } q &\equiv \textbf{choose* } \underline{X} \textbf{ with } \varphi \textbf{ in } \{\,\textbf{skip}; p\,\} \textbf{ ifnone } \{\,\textbf{skip}; q\,\} \\
\textbf{let* } \underline{X} = \underline{t} \textbf{ in } p &\equiv \textbf{choose* } \underline{X} \textbf{ with } \underline{X} = \underline{t} \textbf{ in } p \\
\textbf{let } \underline{X} = \underline{t} \textbf{ in } p &\equiv \textbf{choose } \underline{X} \textbf{ with } \underline{X} = \underline{t} \textbf{ in } p \\
p \vee q &\equiv \textbf{choose* } B\colon \mathbb{B} \textbf{ in if* } B \textbf{ then } p \textbf{ else } q \\
\textbf{if } \varphi \textbf{ then } p \textbf{ else } q &\equiv \textbf{if* } \varphi \textbf{ then } \{\,\textbf{skip}; p\,\} \textbf{ else } \{\,\textbf{skip}; q\,\} \\
\textbf{if } \varphi \textbf{ then } p &\equiv \textbf{if } \varphi \textbf{ then } p \textbf{ else skip} \\
\textbf{abort} &\equiv \textbf{while } \mathit{true} \textbf{ do } \{\,\textbf{skip}\,\}
\end{aligned}$$

may also be used as an abbreviation for the right-hand side.

Note that for the **let** and **let*** abbreviations an additional, omitted renaming of variables is necessary to avoid capture of variables in the expressions $\underline{t}$.

Definition 3.4 (Regular and Sequential Programs)**.** A program that does not use the iteration construct of Def. 3.2 and uses **if** and **choose** instead of **if*** and **choose*** is a *regular* program. A regular program that does not use the fair interleaving is *sequential*.

The semantics of programs is based on the intervals of Sec. 3.1.

Definition 3.5 (Program Semantics)**.** The semantics of programs $I \models [p]_{\underline{X}}$ is given by the largest fixpoint of the derivation system depicted in Fig. 3.1. The variables $\underline{X}$ are the frame assumption of the program. All variables outside of the frame assumption are modified arbitrarily in system steps. The variables of the frame are assigned according to the program.

In following chapters the frame assumption is usually omitted, since all variables of a component with input and output variables are usually used as a frame assumption.

The largest fixpoint is chosen to capture infinite runs of while loops and recursive calls. Note that all rules are productive, i.e., at least one state is added either directly or indirectly (e.g. calls and while) by every rule.

Remark 3.6. In our previous work [120] a shallow embedding of programs into temporal logic is used, by giving a temporal logic formula that defines the semantics of every program construct. Fig. 3.1 depicts a direct, deep embedding of programs with the same semantics. This is more appropriate in the context of this thesis, since its focus are components expressed as programs.

Assignments (3.1) in Fig. 3.1 set the values of the assigned variables $\underline{Y}$ to the values on the left-hand side. All variables $\underline{X} \setminus \underline{Y}$ of the frame assumption are left unchanged and all other variables may change arbitrarily.

The rule for sequential composition of programs (3.2) just reduces to the sequential composition of the individual intervals.

The rule (3.3) for the **choose***-statement chooses a sequence of values $\underline{\sigma}$ for each of the variables $\underline{Y}$ and for each state of the interval. In the initial state the sequence of values $\underline{\sigma}$ must satisfy the condition φ. The changes to the local variables $\underline{Y}$ are therefore invisible in the global interval. If no values that satisfy φ exists, then the **ifnone**-branch is executed.

The rules (3.5) for **if*** either executes the if-branch or the else-branch, depending on the evaluation of the condition φ.

Iteration (3.7) either terminates immediately or unfolds the program once. While loops (3.7) are defined similarly. Fair interleaving (3.10) reduces to fair interleaving of the respective intervals.

A procedure call (3.8) unfolds the definition of the procedure, binds the input variables to local variables and substitutes the reference parameters in the body of the procedure. Note that before and after the body is executed an additional stuttering step is added. These steps

$$\frac{}{I \models [\underline{Y} \;:=\; \underline{e}]_{\underline{X}}} \quad \begin{array}{c} \#I = 1,\ I(0)'(\underline{Y}) = [\![\underline{e}]\!](I(0)), \\ I(0)'(\underline{X} \setminus \underline{Y}) = I(0)(\underline{X} \setminus \underline{Y}) \text{ and} \\ I_b(0) = \texttt{ff} \end{array} \tag{3.1}$$

$$\frac{I_0 \models [p]_{\underline{X}} \quad I_1 \models [q]_{\underline{X}}}{I_0 \fatsemi I_1 \models [p;q]_{\underline{X}}}\ I_1.\mathit{first} = I_0.\mathit{last} \text{ or } \#I_0 = \infty \tag{3.2}$$

$$\frac{I(\underline{Y} \mapsto \underline{\sigma}) \models [p]_{\underline{X},\underline{Y}}}{I \models [\textbf{choose* } \underline{Y} \textbf{ with } \varphi \textbf{ in } p \textbf{ ifnone } q]_{\underline{X}}}\ I(\underline{Y} \mapsto \underline{\sigma}).\mathit{first} \models \varphi \tag{3.3}$$

$$\frac{I \models [q]_{\underline{X}}}{I \models [\textbf{choose* } \underline{Y} \textbf{ with } \varphi \textbf{ in } p \textbf{ ifnone } q]_{\underline{X}}}\ I.\mathit{first} \models \forall \underline{Y}.\, \neg\varphi \tag{3.4}$$

$$\frac{I \models [p]_{\underline{X}}}{I \models [\textbf{if* } \varphi \textbf{ then } p \textbf{ else } q]_{\underline{X}}}\ I(0) \models \varphi \quad \frac{I \models [q]_{\underline{X}}}{I \models [\textbf{if* } \varphi \textbf{ then } p \textbf{ else } q]_{\underline{X}}}\ I(0) \not\models \varphi \tag{3.5}$$

$$\frac{}{I \models [p^*]_{\underline{X}}}\ \#I = 0 \qquad \frac{I \models [p;p^*]_{\underline{X}}}{I \models [p^*]_{\underline{X}}} \tag{3.6}$$

$$\frac{I \models [\textbf{skip}; p; \textbf{while } \varphi \textbf{ do } \{\, p \,\}]_{\underline{X}}}{I \models [\textbf{while } \varphi \textbf{ do } \{\, p \,\}]_{\underline{X}}}\ I(0) \models \varphi \quad \frac{I \models [\textbf{skip}]_{\underline{X}}}{I \models [\textbf{while } \varphi \textbf{ do } \{\, p \,\}]_{\underline{X}}}\ I(0) \not\models \varphi \tag{3.7}$$

$$\frac{I \models [\textbf{let } \underline{Y} = \underline{t} \textbf{ in } p\{\underline{Z} \mapsto \underline{X}\}; \textbf{skip}]_{\underline{X}}}{I \models [\texttt{Proc}(\underline{t}; \underline{X})]_{\underline{X}}}\ \begin{array}{l}\text{Procedure Call} \\ \texttt{Proc}(\underline{Y}; \underline{Z})\, \{\, p \,\}\end{array} \tag{3.8}$$

$$\frac{}{I \models [\textbf{assert } \varphi]_{\underline{X}}}\ \#I = 0, I(0) \models \varphi \qquad \frac{I \models [\textbf{abort}]_{\underline{X}}}{I \models [\textbf{assert } \varphi]_{\underline{X}}}\ I(0) \not\models \varphi \tag{3.9}$$

$$\frac{I_1 \models [\textbf{let* } \mathit{Tid} = 1 \textbf{ in } \{\, p \,\}]_{\underline{X}} \quad \cdots \quad I_n \models [\textbf{let* } \mathit{Tid} = n \textbf{ in } \{\, p \,\}]_{\underline{X}}}{\Big|\Big|_{\mathit{Tid} \in \{1,\ldots,n\}} I_i \models \Big[\Big|\Big|_{\mathit{Tid} \in \{1,\ldots,n\}} p\Big]_{\underline{X}}} \tag{3.10}$$

$$\frac{I_{[1..]} \models [\textbf{atomic } \varphi\ \{\, p \,\}]_{\underline{X}}}{I \models [\textbf{atomic } \varphi\ \{\, p \,\}]_{\underline{X}}}\ \begin{array}{c} I(0) \not\models \varphi, I_b(0) = \texttt{tt}, \\ I(0) = I(0)' \end{array} \tag{3.11}$$

$$\frac{I_0 \models [p]_{\underline{X}}}{I \models [\textbf{atomic } \varphi\ \{\, p \,\}]_{\underline{X}}}\ \begin{array}{c} I(0) \models \varphi, \#I_0 \neq \infty, \#I = 1 \\ I(0) = I_0.\mathit{first}, I_b(0) = \texttt{ff}, I'(0) = I'_0(\#I_0 - 1) \\ I_0 \text{ with empty environment} \end{array} \tag{3.12}$$

$$\frac{I_0 \models [p]_{\underline{X}} \quad I \models [p]_{\underline{X}}}{I \models [\textbf{atomic } \varphi\ \{\, p \,\}]_{\underline{X}}}\ \begin{array}{c} I_0(0) \models \varphi, \#I_0 = \infty, \\ I_0 \text{ with empty} \\ \text{environment} \end{array} \tag{3.13}$$

Figure 3.1: Derivation System for the Program Semantics $I \models [p]_{\underline{X}}$

$$\frac{\Gamma, \psi \vdash [p;q]\,\varphi, \Delta}{\Gamma \vdash [\textbf{atomic}\ \psi\ \{\,p\,\};q]\,\varphi, \Delta} \qquad \frac{\Gamma \vdash \psi, \Delta \quad \Gamma, \psi \vdash (\!|p;q|\!)\,\varphi, \Delta}{\Gamma \vdash (\!|\textbf{atomic}\ \psi\ \{\,p\,\};q|\!)\,\varphi, \Delta}$$

$$\frac{\Gamma, \psi \vdash [p]\,\varphi, \Delta}{\Gamma \vdash [\textbf{assert}\ \psi;p]\,\varphi, \Delta} \qquad \frac{\Gamma \vdash \psi, \Delta \quad \Gamma, \psi \vdash (\!|p|\!)\,\varphi, \Delta}{\Gamma \vdash (\!|\textbf{assert}\ \psi;p|\!)\,\varphi, \Delta}$$

Figure 3.2: Dynamic Logic Rules for Atomic Blocks and Assertions with $(\!|\_|\!)\ \_ \in \{\langle\_\rangle\ \_, \langle\!|\_|\!\rangle\ \_\}$

are used in Ch. 4 as a trigger for invocation and return events. Note that variables might need to be renamed to avoid capture of variables.

An assertion (3.9) either immediately terminates if the condition φ holds, or stutters infinitely.

The semantics of atomic is given by the rules (3.11), (3.12) and (3.13). The idea is that **atomic** $\varphi\ \{\,p\,\}$ blocks until φ is satisfied, then a terminating, sequential run of p is condensed into one system step. If p has non-terminating sequential runs, then the semantics allows arbitrary, runs of p as runs of **atomic** $\varphi\ \{\,p\,\}$ if φ is satisfied. The intuition is that such atomic blocks are essentially ill-used if the inner program has non-terminating runs in the initial state. However, there are several properties of interest for non-termination: First, a non-terminating atomic block should make intermediate states observable, otherwise the atomic block could omit behavior in the event of power failures. Secondly, a non-terminating atomic block should not allow arbitrary behavior.

In order to consider crashes we use the prefix semantics of programs as given by Def. 3.7.

Definition 3.7 (Prefix Semantics)**.** The prefix semantics $[\![p]\!]^{\not\downarrow}$ of program p for some given frame assumption $\underline{X}$ are defined by

$$I \in [\![p]\!]^{\not\downarrow} \Leftrightarrow I \fatsemi I' \models [p]_{\underline{X}} \text{ for some interval } I'.$$

3.3 Dynamic Logic

The three standard modalities from Dynamic Logic [58] $[p]\,\varphi$ (partial correctness), $\langle p\rangle\,\varphi$ (trace existence) and $(\!|p|\!)\,\varphi$ (total correctness) can be defined over the semantics of the previous Sec. 3.2 by considering intervals with empty environment only. Note that only static variables $\underline{x}$ are allowed in the program p of any dynamic logic formula $(\!|p|\!)\,\varphi$. However, in the post-condition arbitrary formulas are permitted.

$$\begin{aligned}
I \models [p]\,\varphi \iff{}& \forall I_0.\ I_0 \text{ finite with empty environment}, I_0(\underline{X}) = I(\underline{x}), I_0 \models [p\{\underline{x} \mapsto \underline{X}\}]_{\underline{X}} \\
&\text{implies } I(\underline{x} \mapsto I_0.last(\underline{X})) \models \varphi \\
I \models \langle p\rangle\,\varphi \iff{}& \exists I_0.\ I_0 \text{ finite with empty environment}, I_0(\underline{X}) = I(\underline{x}), I_0 \models [p\{\underline{x} \mapsto \underline{X}\}]_{\underline{X}} \\
&\text{and } I(\underline{x} \mapsto I_0.last(\underline{X})) \models \varphi \\
I \models (\!|p|\!)\,\varphi \iff{}& \forall I_0.\ I_0 \text{ with empty environment with } I_0(\underline{X}) = I(\underline{x}), I_0 \models [p\{\underline{x} \mapsto \underline{X}\}]_{\underline{X}} \\
&\text{implies } I_0 \text{ finite and } I(\underline{x} \mapsto I_0.last(\underline{X})) \models \varphi
\end{aligned}$$

The variables of the program p are $\underline{x}$ and $\underline{X}$ are fresh, corresponding flexible variables.

Note that the postcondition is evaluated over an interval, namely the interval where the values of the static variables $\underline{x}$ are replaced by their values after executing p. Thus, the

$$\frac{\begin{array}{l}\Gamma^{\underline{x}}_{\underline{X}}, \mathit{rely}(\underline{x},\underline{X}), \mathit{runs}(\underline{X}) \vdash \psi, \Delta^{\underline{x}}_{\underline{X}} \\ \Gamma^{\underline{x}}_{\underline{X}}, \mathit{rely}(\underline{X},\underline{x}) \\ \vdash \langle\!|p\{\underline{X} \mapsto \underline{y}\}|\!\rangle \left(\langle \mathit{rely}(\underline{X}',\underline{X}''),\ \mathit{guar}(\underline{X},\underline{X}'),\ \mathit{inv}(\underline{X}),\ \mathit{runs}(\underline{X}),\ \underline{X} := \underline{y}; q\rangle_{\underline{X}} \varphi\right), \Delta^{\underline{x}}_{\underline{X}}\end{array}}{\Gamma \vdash \langle \mathit{rely}(\underline{X}',\underline{X}''),\ \mathit{guar}(\underline{X},\underline{X}'),\ \mathit{inv}(\underline{X}),\ \mathit{runs}(\underline{X}),\ \textbf{atomic}\ \psi\ \{p\}; q\rangle_{\underline{X}} \varphi, \Delta}$$

$$\frac{\Gamma \vdash \psi, \Delta \quad \Gamma \vdash \langle \mathit{rely}(\underline{X}',\underline{X}''),\ \mathit{guar}(\underline{X},\underline{X}'),\ \mathit{inv}(\underline{X}),\ \mathit{runs}(\underline{X}),\ p\rangle_{\underline{X}} \varphi, \Delta}{\Gamma \vdash \langle \mathit{rely}(\underline{X}',\underline{X}''),\ \mathit{guar}(\underline{X},\underline{X}'),\ \mathit{inv}(\underline{X}),\ \mathit{runs}(\underline{X}),\ \textbf{assert}\ \psi; p\rangle_{\underline{X}} \varphi, \Delta}$$

Figure 3.3: R/G Rules for Atomic Blocks and Assertions

postcondition is not restricted to pure predicate logic, but can again contain dynamic logic formulas, temporal logic formulas and the rely/guarantee formulas introduced in Sec. 3.4.

Assertions and atomic blocks play a key role in the verification methodology of Ch. 5 and Ch. 13. Therefore, Fig. 3.2 depicts the dynamic logic rules for each of the three modalities and the two program constructs. The rules of dynamic logic essentially perform symbolic execution of the program, which transforms the entire program into predicate logic formulas. A proof of the postcondition is then performed in pure predicate logic.

3.4 Rely/Guarantee Reasoning & Calculus

This section introduces rely/guarantee reasoning and presents a calculus for it for the programs of Def. 3.2. Rely/Guarantee reasoning was first introduced by Jones [73] and later on extended by Xu et al. [138] to cover deadlock-freedom and divergence-freedom.

Rely/guarantee reasoning is embedded into the temporal logic of Sec. 3.1 via the R/G formula

$$\left[\mathit{rely}(\underline{X}',\underline{X}''),\ \mathit{guar}(\underline{X},\underline{X}'),\ \mathit{inv}(\underline{X}),\ p\right]_{\underline{X}} \varphi$$

for partial correctness. The formula states that as long as the environment steps satisfy the rely condition $\mathit{rely}(\underline{X}',\underline{X}'')$, the program p satisfies its guarantee condition $\mathit{guar}(\underline{X},\underline{X}')$. Furthermore, the environment as well as the program p propagate the invariant $\mathit{inv}(\underline{X})$. If the program p terminates, then the post-condition φ holds.

The rely/guarantee for (a variant of) total correctness is

$$\langle \mathit{rely}(\underline{X}',\underline{X}''),\ \mathit{guar}(\underline{X},\underline{X}'),\ \mathit{inv}(\underline{X}),\ \mathit{runs}(\underline{X}),\ p\rangle_{\underline{X}} \varphi$$

and ensures, in addition to partial correctness as above, that when all environment steps satisfy the rely condition, then steps of p only block if the runs condition $\mathit{runs}(\underline{X})$ is violated, and that then there are only finitely many non-blocked steps in an execution of p.

If $\mathit{runs}(\underline{X}) \equiv \mathit{false}$ is chosen, then the formula ensures *divergence-freedom* of p, i.e., the property that the program p only performs finitely many non-blocked steps, but might block arbitrarily.

Similar to dynamic logic, the rely/guarantee calculus also performs symbolic execution of the program with additional intermediate proof obligations for the guarantee and invariant. Fig. 3.3 depicts the rules of the R/G calculus for atomic blocks and assertions. For atomic blocks the current assumptions Γ and Δ about flexible variables $\underline{X}$ are replaced with static variables $\underline{x}$ and now refer to the state $\underline{x}$ before arbitrarily many blocked steps. It is only known that these steps were rely steps, since the system steps only blocked and the rely is reflexive and transitive. The atomic block itself is executed in the dynamic logic and calculates a new state $\underline{y}$. The actual step in the interval is performed by the assignment $\underline{X} := \underline{y}$.

$$\frac{\begin{array}{l}\vdash \mathit{rely}(\mathit{tid}, \underline{x}, \underline{x}) \\ \vdash \mathit{rely}(\mathit{tid}, \underline{x}_0, \underline{x}_1) \wedge \mathit{rely}(\mathit{tid}, \underline{x}_1, \underline{x}_2) \rightarrow \mathit{rely}(\mathit{tid}, \underline{x}_0, \underline{x}_2) \\ \Gamma \vdash \mathit{pre}(\mathit{Tid}, \underline{X}), \Delta \\ \vdash \mathit{pre}(\mathit{tid}, \underline{x}_0) \wedge \mathit{rely}(\mathit{tid}, \underline{x}_0, \underline{x}_1) \rightarrow \mathit{pre}(\mathit{tid}, \underline{x}_1) \\ \vdash \mathit{tid}_0 \neq \mathit{tid}_1 \wedge \mathit{guar}(\mathit{tid}_0, \underline{x}_0, \underline{x}_1) \rightarrow \mathit{rely}(\mathit{tid}_1, \underline{x}_0, \underline{x}_1) \\ \mathit{pre}(\mathit{Tid}, \underline{X}) \vdash \langle \mathit{rely}(\mathit{Tid}, \underline{X}', \underline{X}''),\ \mathit{guar}(\mathit{Tid}, \underline{X}, \underline{X}'),\ \mathit{inv}(\underline{X}),\ \mathit{false},\ p\rangle_{\underline{X}}\ \mathit{true}\end{array}}{\Gamma \vdash \langle \underline{X}' = \underline{X}'',\ \mathit{true},\ \mathit{inv}(\underline{X}),\ \mathit{false},\ \Big\|_{\mathit{Tid} \in \{1,\ldots,n\}} p\rangle_{\underline{X}}\ \mathit{true}, \Delta}$$

Figure 3.4: R/G Decomposition Rule for a Concurrent System

The rule facilitates the use and reuse of lemmas about (sub)programs expressed in dynamic logic, which are by far easier to prove.

The rule for assertions in the figure just ensures that the assertion holds before the program p is executed.

In order to decompose a parallel system with an empty environment into proof obligations for individual threads, the rule depicted in Fig. 3.4 is used. The rule ensures that the system is *divergence-free*. The premises state that the rely condition must be reflexive and transitive, the precondition must hold initially and must be stable over steps of other threads, the guarantee condition of one thread must imply the rely of any other thread, and the program p itself sustains the guarantee and invariant. In Ch. 4 we will only show divergence-freedom during invariant proofs. In Ch. 13 Lipton reductions will ensure that the entire system is deadlock-free.

4 Concurrent & Sequential, Crash-Aware Components

Summary. This Chapter introduces a specification mechanism for concurrent and sequential components that are aware of and affected by power cuts or other fatal, system-wide crashes. The components are state-based. The state is encapsulated and hidden from the clients of the component. A correctness criterion for crash-aware components with respect to a sequential specification is given. The criterion facilitates refinement of components and admits component substitution to construct large crash-aware systems, such as file systems, in a modular fashion.

Publications. This Chapter is based on the publications [109, 45, 47, 107]. It extends the semantics that we give in [107] to concurrency and adds internal operations with guards.

Contents

4.1 Crash-Aware Components

This Chapter introduces a formalism that facilitates developing large systems that need to provide guarantees in the event of a power cut or other fatal crashes.

File systems as well as database management systems are an example of this. They both need to ensure that a consistent state is recovered after a power failure or system crash. For performance, however, these systems routinely cache parts of the data. Only eventually the data is persisted to the storage medium. These systems therefore are not able to recover the state prior to a crash completely. From a verification perspective it is therefore necessary to be able to express what exactly happens in the event of a power failure, i.e., which information exactly is lost. This is not only a problem of expressive power, it is also important that the employed specification mechanism is understandable to users and that it makes verification tractable or at least facilitates it. The effect of a power failure is modeled as two distinct parts: First the effect of the crash is applied, and afterwards the component's recovery operation is run.

A second aspect that is crucial for the development of large systems, is that they can be composed in a modular fashion from smaller building blocks: We want to abstract from a part A of the system and only use a (much simpler) specification A^{Spec} of A for the verification of a property P of another part M that uses A. We express the notion that M uses A as $M \multimap\!\!\copyright\!\!\multimap A$. Compositionality of the verification method means that if

1. A^{Spec} is a specification of A, written $A^{Spec} \sqsubseteq A$ (A refines A^{Spec}), and
2. $M \multimap\!\!\copyright\!\!\multimap A^{Spec}$ has property φ

then the actual system of interest $M \multimap\!\!\copyright\!\!\multimap A$ has property φ, too.

We use *components* in order to *implement* system parts such as A and M and to *specify* them, i.e., A^{Spec} is also expressed as a component. Composing several components again yields another component. Thus, components are used in this thesis as an implementation and specification mechanism for individual parts of a large systems and for a composition of such parts.

The remainder of this section defines the syntax of components. Then in Sec. 4.2 several small examples are discussed that highlight the two specification mechanisms for crashes. Section 4.3 presents the semantics of a component in detail. Section 4.4 defines the correctness criterion we prove for components and Sec. 4.5 ensures that a specification component can be replaced by a correct implementation component. Finally, Sec. 4.6 shows how we ensure that invariants hold for a component and its subcomponents.

A component consists of several interface and internal operations over a common state. The state itself is hidden from clients.

Definition 4.1 (Interface and Internal Operations)**.** An *interface operation*

$$\texttt{Op}(\underline{in}\colon \underline{\texttt{In}}; \underline{x}\colon \underline{\texttt{St}}, \underline{out}\colon \underline{\texttt{Out}})\ \textbf{pre}\ \varphi(\underline{in}, \underline{x})\ \{\, p \,\}$$

over the state variables $\underline{x}$ consists of a precondition given as a formula φ and an implementation p that takes the variables $\underline{in}$ of types $\underline{\texttt{In}}$ and the state $\underline{x}$ as input and produces an output in the variables $\underline{out}$ of types $\underline{\texttt{Out}}$ and updates the state $\underline{x}$. An internal operation

$$\texttt{IOp}(;\underline{x}\colon \underline{\texttt{St}})\ \textbf{guard}\ \varphi(\underline{x})\ \{\, p \,\}$$

consists of a guard φ and a program p that is triggered only if the guard is satisfied.

Interface as well as internal operations are just procedures with preconditions and guards. The program that corresponds to the interface operation $\texttt{Op}$ is $\{\, \textbf{assert}\ \varphi(\underline{in}, \underline{x}); p \,\}$. For the internal operation the program is $\{\, \textbf{if*}\ \texttt{guard}(\underline{x})\ \textbf{then}\ p \,\}$. Preconditions and guards are only treated specially in the semantics of Sec. 4.3, the verification in Sec. 4.6 will always ensure that preconditions are met by the caller.

Definition 4.2 (Crash-Aware Component)**.** A crash-aware component C is a tuple

$$C = \left(\underline{x}^C\colon \underline{\texttt{St}}^C, \texttt{init}^C, \{\texttt{Op}_i^C\}_{i\in\mathfrak{I}}, \{\texttt{IOp}_k^C\}_{k\in\mathfrak{K}}, sync^C, crash^C, \texttt{recover}^C, \{C_l\}_{l\in\mathfrak{L}}\right).$$

consisting of

- state variables $\underline{x}^C$ of types $\underline{\texttt{St}}^C$,
- an initialization operation $\texttt{init}^C$,
- interface operations $\texttt{Op}_i^C$ for $i \in \mathfrak{I}$,
- internal operations $\texttt{IOp}_k^C$ for $k \in \mathfrak{K}$,
- a predicate $sync(\underline{x}^C)$ characterizing synchronized states,

- a predicate $crash(\underline{x}_0^C, \underline{x}_1^C)$ characterizing the effect of a power cut,
- a recovery operation `recover` that reconstructs a desirable state after a crash, and
- a (possibly empty) set of direct subcomponents C_l with a state disjoint from $\underline{x}$ and pairwise disjoint from each other.

The program associated with all operations must be regular and sequential. All operations operate over and preconditions are defined over the combined state space of C and all its direct subcomponents C_l (and so on recursively). However, guards may only refer to the state variables $\underline{x}^C$.

Notation ($\_^C$ and $\widehat{\_}^C$). We refer to the state, operations, predicate and subcomponents of a component C by adding a superscript $\_^C$, if the referred component C is not clear from the context. For example $\texttt{init}^C$ refers to the initialization operation of component C.

We refer to the variables and predicates of the combination of the component C with its subcomponents (recursively) by $\widehat{\_}^C$. For example $\widehat{\underline{x}}^C$ refers to the disjoint union of the state variables of component C and all of its subcomponents. For predicates p the notation $\widehat{p}^C$ refers to the logical conjunction of the individual predicates of the component C and all its subcomponents. If the component is clear from context we write $\widehat{\_}$.

A component can be thought of an extension of standard data types [60, 40] and concurrent objects [62] with crash behavior and a recovery operation. It provides an interface of operations with input and output to a client. Before using a component the initialization operation needs to be called. In between calls to interface operations the component might trigger internal operations when their guard is satisfied. An example for such an operation that needs to be executed in the background is garbage collection and wear-level in the context of a file system for flash memory.

At every point in the execution of an operation a power cut or fatal crash might happen. The immediate effect is described by the two predicates *sync* and *crash*: The basic idea is that *sync* describes *synchronized states*, which are almost immune to power failures. In the event of a power failure the component is then taken back to a previous synchronized state, which rescinds or *retracts* several of the operations before the power failure. Afterwards only a residual effect characterized by the predicate *crash* is visible. We allow that the predicate *crash* is partial, which means that a power failure only needs to be considered in or is observable in a desirable and more easily understood set of states. At a later point when the system restarts the component has a chance to recover its internal state via the `recover` operation. Afterwards the component can again be used by a client calling its operations. The exact workings of a power failure and the relationship between the two predicates are discussed in more detail in Sec. 4.3.

In order to perform its task a component C may use its subcomponents C_l. A component C and its subcomponents C_l are in the following visualized as shown in Fig. 4.1. The notation is similar to UML component diagrams [5]: The gripper symbol ⊣(denotes the required interface of the component C and the lollipop symbol o— denotes the provided interface of C_l. Both interfaces may be connected if they match, denoted by the combined symbol —◎— with the restriction detailed in Def. 4.3 on what constitutes a weakly admissible composition of components. A stronger version of admissibility that admits substitution of the subcomponents C_l is defined in Sec. 4.5. The composition then provides the interface of C.

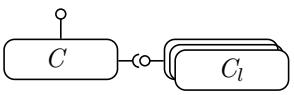

Figure 4.1: Component C with Subcomponents C_l

The component hierarchy forms a tree, because every subcomponent is part of the surrounding component. Sharing a subcomponent across several client components is therefore not possible. This would also require showing that the two clients' use of the subcomponent do not interfere with each other. In contrast to object-oriented design the component hierarchy is statically known. It is not possible to add additional instances of components during the system's runtime.

Definition 4.3 (Weak Admissibility)**.** A component C is *weakly admissible* if and only if

1. no state variable of any subcomponent C_l is read from or written to directly in any of the operations of C and vice versa (information hiding),
2. operations of C only call interface operations of its subcomponents or other auxiliary operations of C,
3. states after initialization and recovery must be synchronized states,
4. initialization and recovery of component C must call the corresponding operations of all its subcomponents before any of their interface operations,
5. the synchronized states must be in the domain of the (potentially partial) crash predicate, and
6. each subcomponent C_l is weakly admissible for each $l \in \mathfrak{L}$.

All of these properties except (3.) and (5.) can be checked statically. The property (3.) is ensured by the invariant proofs of Sec. 4.6. For (5.) a simple predicate logic proof obligation suffices.

If access to a state variable of a subcomponent is desired, then an operation providing such access is required. Indirect access to the state of subcomponents therefore is possible. If access to the state of a surrounding component is required, the state variable must be passed as a regular input/output parameter to the subcomponent's operation. Encapsulating the state in this form is necessary to be able to replace components. Otherwise, substitution of components depends on assignments to the state variables from the context (in this case the surrounding component).

Note that Def. 4.3 implicitly prohibits callbacks, since C cannot call operations from its surrounding component. Therefore, C_l cannot call C either.

We distinguish *retracting* from *non-retracting* components, only the latter make use of the specification mechanism of synchronized states. The naming will be more apparent once the semantics is discussed in more detail in Sec. 4.3.

Definition 4.4 ((Non-)Retracting Components)**.** A component C is *non-retracting* if the synchronized predicate $sync^C$ is equivalent to *true*. A component is called *retracting* otherwise.

Implementations are always non-retracting components and must allow power failures in any intermediate state.

Definition 4.5 (Implementation Component)**.** A component C is an *implementation* if and only if it is non-retracting and the crash predicate $crash^C$ is total.

Specifications on the other hand must be atomic and may never block.

Definition 4.6 (Specification Component)**.** A component A is a *specification* if and only if each of its interface and internal operations consists of an atomic block, where the guard of the atomic block is equivalent to *true*, and always terminates within the precondition.

The invariant proofs in Sec. 4.6 will ensure that a component is indeed a specification component.

An easy way of understanding the crash predicate of implementation components C is considering whether a state variable is modeled as if it resides in volatile memory. A state variable therefore is part of the RAM state if and only if it may have an arbitrary value after a crash and before the recovery.

component *Counter*
  subcomponent *Storage*
state
  cnt: $\mathbb{N}$
initialization
  `init`(*n*: $\mathbb{N}$)
    pre $\varphi(cnt)$
    { *Storage*.`init`(n); $cnt \coloneqq n$ }
interface operations
  `inc`() { $cnt \coloneqq cnt + 1$ }
  `get`(; *n*: $\mathbb{N}$) { $n \coloneqq cnt$ }
internal operations
  `persist`()
    guard $\varphi(cnt)$
    { *Storage*.`store`(*cnt*) }
synchronized states
  true
crash
  true
recovery
  `recover`() {
    Storage.`recover`()
    Storage.`load`(; *cnt*)
  }

Figure 4.2: Counter Component

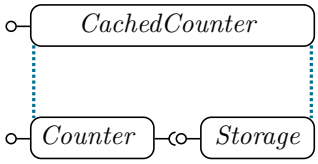

Figure 4.3: Cached Counter Component Structure

component *Storage*
state
  pcnt: $\mathbb{N}$
initialization
  `init`(*n*: $\mathbb{N}$) { $pcnt \coloneqq n$ }
interface operations
  `store`(*n*: $\mathbb{N}$) { $pcnt \coloneqq n$ }
  `load`(; *n*: $\mathbb{N}$) { $n \coloneqq pcnt$ }
synchronized states
  true
crash
  $pcnt' = pcnt$
recovery
  `recover`() { **skip** }

Figure 4.4: Storage Component

Definition 4.7 (RAM State of an Implementation)**.** A state variable x_i^C is part of the *RAM state* of an implementation component C with state variables $\underline{x}^C = \langle x_1^C, \ldots, x_n^C \rangle$ if and only if

$$\forall \underline{x}^C, \underline{y}^C.\ crash^C(\underline{x}^C, \underline{y}^C) \rightarrow \forall z_i^C.\ crash^C(\underline{x}^C, y_0^C, \ldots, y_{i-1}^C, z_i^C, y_{i+1}^C, \ldots, y_n^C)$$

holds.

Definition 4.8 (RAM Component)**.** An implementation component C is a *RAM component* if and only if the RAM state of C comprises the entire state of C.

Note that a component C is a RAM component if and only if the crash predicate $crash^C$ is equivalent to *true*. A RAM component may contain a subcomponent that is not entirely in RAM.

4.2 A Cached Counter as an Example

Before delving into the semantics of components in Sec. 4.3 this section shows several simple sequential components as examples to further intuition. Fig. 4.3 depicts a sequential system consisting of a component *Counter* that implements a counter that is persisted in the component *Storage* in the background. The component *CachedCounter* is an abstraction of the system, i.e., *Counter*—©—*Storage* implements or *refines* the specification *CachedCounter*. Refinement is indicated with the two **blue dotted** lines.

Storage The specification of the persistent storage is shown by Fig. 4.4. It features a state variable *pcnt* for the value of the persisted counter. Two interface operations `store` and `load`

are provided in order to access the stored counter. Initially, the persistent counter is set to a given value. The effect of a power cut is characterized by the crash predicate

crash
$pcnt' = pcnt$

which relates the state immediately after the power cut, denoted by $pcnt'$, to the state $pcnt$ right before the power cut. In this case the counter is unchanged by a power cut, which is characteristic for persistent data. Recovery after a power cut then has to restore or repair nothing.

The following component specification will just omit recover and initialization operations with the body **skip** and predicates that are *true*, such as the synchronized predicate of *Storage*.

Counter The component *Counter* of Fig. 4.2 uses the storage component and augments it with a cached version of the counter (state *cnt*). It provides an increment operation `inc`, which only increments the cached counter, and an operation `get` to read the value of the cached counter. In the background (internal operation `persist`) the component persists the cached counter to storage from time to time. On a power cut the value of the cached counter is lost, i.e., set to an arbitrary value and the recovery has to restore the value from the component *Storage*.

It might be useful or necessary to hold back on persisting a value until a condition $\varphi(cnt)$, shown in **red** in Fig. 4.2, is satisfied. This could increase performance or it might be inevitable due to hardware limitations. For hard disks and flash memory writing is only allowed at a certain granularity, namely at the level of sectors and pages, respectively. Note that the precondition or guard does not imply that *every* value of *cnt* that satisfies $\varphi(cnt)$ is persisted. It just means that if `persist` is triggered then $\varphi(cnt)$ is satisfied. If initially $\varphi(cnt)$ also holds then after every power cut we recover a state where $\varphi(cnt)$ also holds.

The component *Storage* obviously is an *abstract* specification of a storage device. A real implementation would need to serialize the counter to some byte representation before storing it and deserialize the counter value from its byte representation after loading it. However, this shows the strength of an approach that uses the same mechanism for specification and implementation. We can later on provide another component that implements the same functionality as *Storage* based on a real, byte-based storage medium and plug this implementation into the component *Counter* without affecting its functionality or jeopardizing its correctness.

CachedCounter (state-based) One possible way to abstract both components is shown by the component *CachedCounter* in Fig. 4.5. The interface is the same as that of the *Counter* with an operation for incrementing and reading the cached counter. The state collapses both counters of the implementation into one state variable (code in **black** only). Whether such a simplification of the state is adequate depends on what guarantees a client of the *CachedCounter* expects in the event of a power failure.

If for example the client does not care at all what value the counter has after a crash, a crash predicate of *true* is obviously sufficient. This would not even guarantee that the value afterwards is in between the initial value of the counter and the value right before the power failure. If on the other hand it is sufficient that the value after a crash is not greater than before, the crash predicate given in Fig. 4.5 (**black** part only) can be used. The guarantee that the counter after a power failure is in between its initial and last value requires the additional state variable *icnt* for the initial value of the counter and the code depicted in **green** to keep track of it.

component *CachedCounter*
state
$ccnt$: $\mathbb{N}$, $icnt$: $\mathbb{N}$
initialization
`init`(n: $\mathbb{N}$)
pre $\varphi(ccnt)$
{ $ccnt \coloneqq n$, $icnt \coloneqq n$ }
interface operations
`inc`() { $ccnt \coloneqq ccnt + 1$ }
`get`(; n: $\mathbb{N}$) { $n \coloneqq ccnt$ }
crash
$icnt \leq ccnt' \leq ccnt$
$\wedge\ icnt' = ccnt'$
$\wedge\ \varphi(ccnt')$

Figure 4.5: Cached Counter Component (State-based)

component *CachedCounter*
state
$ccnt$: $\mathbb{N}$, $synced$: $\mathbb{B}$
initialization
`init`(n: $\mathbb{N}$)
pre $\varphi(ccnt)$
{ $ccnt \coloneqq n$, $synced \coloneqq true$ }
interface operations
`inc`() { $ccnt \coloneqq ccnt + 1$, $synced \coloneqq false$ }
`get`(; n: $\mathbb{N}$) { $n \coloneqq ccnt$, $synced \coloneqq false$ }
internal operations
`persist`()
guard $\varphi(ccnt)$
{ $synced \coloneqq true$ }
synchronized states
$synced$
crash
$ccnt' = ccnt \wedge synced \wedge \varphi(ccnt)$

Figure 4.6: Cached Counter Component (Operations-based)

If the precondition $\varphi(cnt)$ of the internal operation `persist` in Fig. 4.2 is used, then the abstraction can also ensure that a crash yields a state where this condition holds, as shown in **red** in Fig. 4.5.

The approach we take here to specifying the behavior on a power cut is *state-based*, i.e., we try to express the state transition undertaken during a crash by the state before and after the crash. What is usually lost is how the state reached afterwards relates to the history of the system. A second concern is that such a specification is usually quite difficult to understand in a realistic setting with more complex components. Finally, in a verification such an effect must be propagated upwards a component hierarchy *explicitly* at every step. For these reasons, we now look at a different, more abstract method of specification for the example component.

CachedCounter (operations-based) Another approach to specify a crash is in terms of the operations executed before the power failure. A component may *retract* several of the last operations up until a *synchronized state* is encountered, which is then used as the state after the power failure. The idea is not that the system actively rolls back an operation, which is usually associated with aborting a transaction of a database system, rather the state after the power failure *looks as if* some operations did not take effect before the power failure in the first place. This gives an alternative run of the component with fewer operations that explain the state of the component after a power failure. This view on the effect of a power failure is termed *operations-based* in the following.

Fig. 4.6 uses this specification paradigm for the cached counter. An additional state variable $synced$ is used to characterize synchronized states that are written to storage by the internal operation `persist`. This guarantees that in the event of a power failure several of the last executed operations are retracted but never persisted. The crash predicate then leaves $ccnt$ unaltered, since the entire effect of a power failure is captured by the synchronized states.

Again, if the implementation persists the counter only if $\varphi(cnt)$ holds, i.e., if the operation `persist` has guard $\varphi(cnt)$ as in the figure, then additionally in Fig. 4.6 we may assume that the crash predicate is only applied in states where $\varphi(ccnt)$ holds.

A critical part of the semantics is that both specification mechanism are supported via the **crash** and **synchronized** predicates. For implementations the state-based view is natural, because all RAM-state is just assigned arbitrarily and reversing the effect of operations does not correspond to its actual behavior. However, for a specification the state-based view usually has the drawback that a lot of auxiliary state is needed to express the power cut suitably. This impedes the understandability of specifications and hampers the verification of clients. There an operations-based view is more convenient and under certain conditions the operations-based view can be propagated upwards a component hierarchy implicitly, i.e., with only minimal effort during verification.

4.3 Semantics of Components

We use standard notations from the concurrency literature [62] to express the behavior and runs of concurrent and sequential crash-aware components. Histories capture the input and output behavior of a component and are extended with reset events. A *reset* describes the entire process of the retraction of several operations, and the crash transition with subsequent recovery.

Definition 4.9 (Events & Histories)**.** A *history* h is a finite sequence of events. An *event* is either

1. an invocation event $inv^{C}_{Tid}(j, \underline{in})$ of the operation j of the component C by thread Tid with input values $\underline{in}$, or
2. a return event $ret^{C}_{Tid}(j, \underline{out})$ of the operation j of the component C by thread Tid with output values $\underline{out}$, or
3. a reset event *reset*

where j is a valid index for internal or interface operations of the respective component C. The input and output values must match the type of input and output variables of operation $\texttt{Op}^{C}_{j}$. A history is *reset-free* if it does not contain a reset event.

Definition 4.10 (Sequential History)**.** A history h is *sequential* if and only if

- the first event is an invocation, and
- every invocation $inv^{C}_{Tid}(j, \underline{in})$ is followed by a matching return event $ret^{C}_{Tid}(j, \underline{out})$ for some output $\underline{out}$ or a reset event, or it is the last event of the history, and
- return events are always preceded by a matching invocation event.

We denote with $h\lfloor Tid$ the sequence of events of thread Tid including reset events. For two histories h_0 and h_1, $h_0 \cdot h_1$ denotes their concatenation.

Definition 4.11 (Well-formed History)**.** A history h is *well-formed* if and only if $h\lfloor Tid$ is sequential for every thread Tid.

All histories considered in this thesis are well-formed since well-formed histories arise naturally from standard program semantics, where one thread can only execute one operation at any time.

Definition 4.12 (Completed Histories)**.** The completed part of a reset-free history h, denoted by $completed(h)$ is the sequence of all invocations with matching return events in h.

Definition 4.13 (Pending Operations)**.** An execution of an operation is pending in a reset-free history h if there is no matching return event for the invocation event.

Definition 4.14 (Real-Time Order)**.** The real-time order $<_h$ of operation *executions* op and op' induced by a reset-free history h is defined by

$$op <_h op' \quad \text{iff} \quad \text{the return event of } op \text{ precedes the invocation of } op' \text{ in } h.$$

Two operation executions not ordered by the real-time order $<_h$ are executed *concurrently* in h.

Definition 4.15 (Linearization)**.** A reset-free history h is *linearizable* if it is possible to extend h with return events to a history h' and *completed*(h') is equivalent to a well-formed, sequential history h'' that respects the real-time order of events of h', i.e., $<_h \subseteq <_{h''}$ holds. The history h'' is called a *linearization of* h.

Respecting the real-time order of events means that only operations with overlapping execution may be re-ordered by its linearization. Equivalent histories contain the same invocation and return events with the same inputs and outputs.

Def. 4.16 extends the program semantics of Ch. 3 such that histories of calls to interface and internal operations are generated, see our previous work [47] for details about the extended derivation system.

Definition 4.16 (Program Semantics with Histories)**.** The judgment $I, h \models p$ extends $I \models p$ with the history generated by the program p, i.e., it records invocation and return events of calls to interface and internal operations of each component. We associate the invocation event of a call to $\mathtt{Op}$ with the first, stuttering step of the call and the return event with the last, stuttering step of the call.

Histories h with $I, h \models p$ correspond to the observable behavior of the execution I of the program p. Given an interval and a history with $I, h \models p$, we write $I|_n$ and $h|_n$ for the interval and corresponding history of the first n system steps.[1]

The state space S of the component

$$C = \left(\underline{x}^C : \underline{\mathtt{St}}^C, \mathtt{init}^C, \{\mathtt{Op}_i^C\}_{i\in\mathfrak{I}}, \{\mathtt{IOp}_k^C\}_{k\in\mathfrak{K}}, sync^C, crash^C, \mathtt{recover}^C, \{C_l\}_{l\in\mathfrak{L}}\right).$$

is given by the cartesian product of the carrier sets (in some given algebra) for the types $\widehat{\underline{\mathtt{St}}}^C$ of all state variables.

First, interface operation steps and internal operation steps are defined by Def. 4.17 and Def. 4.18.

Definition 4.17 (Interface Operation Execution)**.** An interval I with history h is an *interface operation execution* of the component C, written $I, h \models Op^C$, if there is an interface operation $\mathtt{Op}_i$ with $i \in \mathfrak{I}$ and with declaration

$$\mathtt{Op}_i(\underline{in}; \widehat{\underline{x}}^C, \underline{out}) \textbf{ pre } \varphi(\underline{in}, \widehat{\underline{x}}^C) \ \{\, p \,\}$$

and either

- $I(0) \models \varphi(\underline{in}, \widehat{\underline{x}}^C)$ and $I, h \models \mathtt{Op}_i(\underline{in}; \widehat{\underline{x}}^C, \underline{out})$, or
- $I(0) \not\models \varphi(\underline{in}, \widehat{\underline{x}}^C)$ and the interval I is arbitrary for the state variables $\underline{x}$ of component C and the history contains the corresponding invoke and response events, i.e., the first event is always the event $inv^C(I(0)(\underline{in}), I(0)(\widehat{\underline{x}}^C))$ and if the interval is finite then the second and final event is the corresponding return event for the operation

holds.

[1] Note that technically it is necessary to add additional τ events to the history in order to be able to split off the corresponding prefix of the history, however, we will omit this technical detail in the following.

Note that in the case of Def. 4.17 where the precondition holds the history h automatically records the corresponding events of the call according to Def. 4.16. The semantics of procedure calls (Fig. 3.1 on page 21) with one additional stuttering step in between the time the event is added to the history and the body of the procedure is executed, ensure that it is always possible for a reset event to occur directly after the invocation event and directly before the return event, even if the body of the procedure is already atomic.

Definition 4.18 (Internal Operation Execution)**.** An interval I with history h is an *internal operation execution* of the component C, written $I, h \models \widehat{IOp}^{C}$, if there is an internal operation `IOp` of the component C or any of its (recursive) subcomponents with

$$\texttt{IOp}_k(;\widetilde{\underline{x}}) \textbf{ guard } \varphi(\widetilde{\underline{x}}) \ \{\, p \,\}$$

for the state $\widetilde{\underline{x}}$ of the corresponding (sub)component and

$$I(0) \models \varphi(\widetilde{\underline{x}}) \text{ and } I, h \models \texttt{IOp}(;\widetilde{\underline{x}})$$

holds.

Def. 4.17 and Def. 4.18 correspond to the intuition that if a precondition of an interface operation is violated then arbitrary behavior may follow and that internal operations are only triggered if their guard is satisfied.

Notation. In the following intervals with stuttering environment steps and unblocked system steps are written as a sequence of states $(s_0, \ldots, s_n)$, i.e., the stuttering environment steps and the blocked flag are omitted. As a shorthand for the invoke and return events of a sequential component we write $s_0 \overset{\texttt{Op}(in,out)}{\rightsquigarrow} s_n$ for a complete operation execution starting in state s_0 with input values in, finishing in state s_n with output values out. This is equivalent to $(s_0, \ldots, s_n) \models \texttt{Op}(\underline{in}; \widehat{\underline{x}}, \underline{out})$ where $\widehat{\underline{x}}$ is the (hidden) state of the corresponding component. If the operation is atomic we use a straight arrow as in $s_0 \xrightarrow{\texttt{Op}(in,out)} s_1$. We write $s_0 \overset{\texttt{Op}(in,\lightning)}{\rightsquigarrow} s_n$ if the operation is interrupted by a power cut in an intermediate state s_n of its execution and a reset occurs afterwards. Then no actual output is visible, i.e., the return event is missing and the invocation event is followed by a *reset* event and a Reset transition.

For sequential crash-aware components the system $System \equiv System^{seq}$ is executed and for concurrent crash-aware components the system $System \equiv System^{con}$ is chosen as defined by equations (System).

$$System^{seq} \equiv \{\, \textbf{choose*}\ in, out\ \textbf{in}\ \{\, Op \vee \widehat{IOp}^{C} \,\}\ \}^{*} \tag{System}$$

$$System^{con} \equiv \bigparallel_{Tid \in \{1,\ldots,n\}} \{\, \textbf{choose*}\ in, out\ \textbf{in}\ \{\, Op \vee \widehat{IOp}^{C} \,\}\ \}^{*}$$

The semantics of crash-aware components is the set of its runs. Each run consists of an interval I and the corresponding history h, which are the result of executing $System$ with intermittent reset transitions.

Definition 4.19 (Semantics of Crash-Aware Components)**.** An interval I with empty environment over the variables $\widehat{\underline{x}}^{C}$ and with corresponding history h is a run of component C, written $(I, h) \in runs(C)$, if I and h satisfy the following properties.

1. There is an interval I' satisfying $I' \models \texttt{init}(;\widehat{\underline{x}}^{C})$ and $I.first = I'.last$
2. There exist finitely many intervals with corresponding histories

   $$(\widetilde{I}_0, \widetilde{h}_0), \ldots, (\widetilde{I}_n, \widetilde{h}_n) \qquad \text{with} \qquad (\widetilde{I}_j, \widetilde{h}_j) \models System \qquad \text{for all } 0 \le j \le n.$$

   Each interval $\widetilde{I}_j$ and history $\widetilde{h}_j$ corresponds to a complete execution of the system.

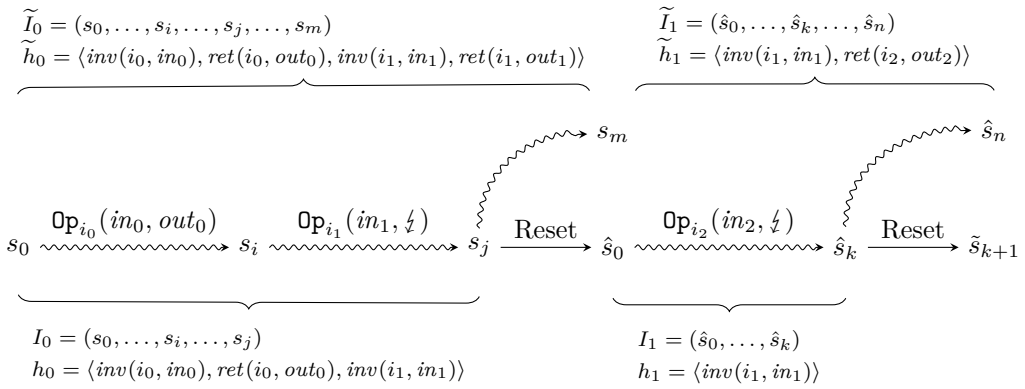

Figure 4.7: Example Run with two Power Failures

3. For all except the last interval there is a number of system steps n_i for $0 \leq j < n$ that are actually executed up to the reset transition. We denote with $I_j \equiv \widetilde{I}_j|_{n_j}$ and $h_j \equiv \widetilde{h}_j|_{n_j}$ these system steps with their corresponding history for $0 \leq j < n$.

4. It is possible to construct a *reset transition* $I_j.last \xrightarrow{\text{Reset}} I_{j+1}.first$ *based on* I_j *and* h_j for each j as given by Def. 4.20 below.

The interval I is then the sequential composition

$$I = I_0 \fatsemi (I_0.last, I_1.first) \fatsemi I_1 \fatsemi \ldots \fatsemi I_{n-1} \fatsemi (I_{n-1}.last, I_n.first) \fatsemi \widetilde{I}_n$$

with one intermediate system step for the reset transition between the individual intervals I_j. The history h is the sequential composition

$$h = h_0 \cdot \langle reset\rangle \cdot h_1 \cdot \ldots \cdot h_{n-1} \cdot \langle reset\rangle \cdot h_n$$

with reset events in between the individual history segments h_j.

Fig. 4.7 shows an example run of a sequential crash-aware component with two power failures, interrupting the operations $\mathtt{Op}_{i_1}$ and $\mathtt{Op}_{i_2}$ in intermediate states. The figure also shows the intervals and histories of Def. 4.19. The intervals $\widetilde{I}_j$ and histories $\widetilde{h}_j$ are depicted at the top and the final intervals I_j and histories h_j at the bottom. The complete execution of the operations $\mathtt{Op}_{i_0}$ and $\mathtt{Op}_{i_1}$ consists of the interval $\widetilde{I}_0$ with corresponding history $\widetilde{h}_0$. In the figure this execution is interrupted by a power failure in state s_j. The reset transition then starts in state s_j and yields a new state $\hat{s}_0$. The third operation $\mathtt{Op}_{i+2}$ is also interrupted in an intermediate state $\hat{s}_k$. The observable behavior, i.e., the history, of the entire run is therefore

$$h = h_0 \cdot \langle reset\rangle \cdot h_1 \cdot \langle reset\rangle,$$

which contains an invocation for all three operations, but a return event for the first operation, only.

We point out some aspects of Def. 4.19 before formally defining reset transitions.

Power cuts during the initialization of a component are not captured by this semantics. The reasoning is that crashes in intermediate states of the initialization are not really of any interest. Recovery in such a state is not possible and initialization, which usually formats the storage device in a file system, has to be run again, which must be triggered explicitly by the user.

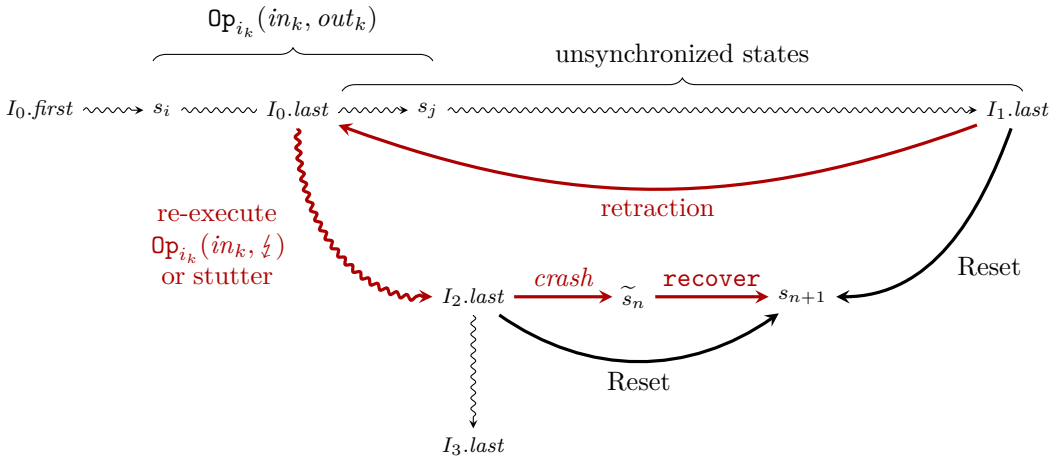

Figure 4.8: Constructing a Reset Transition and an Alternative Execution for a Sequential Component with only one Pending Operations

A non-terminating execution of an operation is recognized in a run according to Def. 4.19 if the history has an invocation event without a matching return event or an interrupting reset event.

The reset transitions are given by Def. 4.20, which should be followed alongside Fig. 4.8. Note that the figure again depicts the sequential case, where there is only one pending operation at any time. The general idea is that we first revert several operations and then allow a re-execution of pending operations until a state is reached where the partial predicate *crash* and recovery are applied. Retractions may not cross synchronizes states, though.

Definition 4.20 (Reset Transitions)**.** A reset transition $I.last \xrightarrow{\text{Reset}} s_{n+1}$ of component C based on interval $I = (s_0, \ldots, s_n)$ with corresponding history h is constructed in four steps visualized by the four **red transitions** in Fig. 4.8.

1. The original execution I is split into two intervals $I_0 \,{}^{\circ}_{9}\, I_1$ with corresponding histories $h_0 \cdot h_1$ and with the property that I_1 does not contain any synchronized states after its system steps, i.e., $I_1(k)' \not\models \widehat{sync}(\widehat{x})$ holds for all k with $0 \leq k \leq \#\, I_1$. All steps of I_1 are retracted.

2. The system may re-execute part of the operations pending in h_0 and crash afterwards, i.e., there exist intervals I_2 and I_3 and corresponding histories h_2 and h_3 with

   $$I_0 \,{}^{\circ}_{9}\, I_2 \,{}^{\circ}_{9}\, I_3, h_0 \cdot h_2 \cdot h_3 \models \mathit{System} \qquad \text{and} \qquad I_2.last \models \exists \underline{\widehat{x}}_1.\ \widehat{crash}(\widehat{\underline{x}}, \underline{\widehat{x}}_1)$$

   and history h_2 only contains return events (for pending invocations in h_0). The interval I_3 and corresponding history h_3 are removed, i.e., interval I_2 and history h_2 are a partial or complete (re-)execution of all pending operations of the component.

3. The effect of the crash is applied to the state $I_2.last$ and yields a state $\widetilde{s}_n$, i.e., a state $\widetilde{s}_n$ must exist with $(I_2.last, \widetilde{s}_n) \models \widehat{crash}(\widehat{\underline{x}}, \widehat{\underline{x}}')$.

4. The recovery routine reconstructs the final, synchronized state s_{n+1}, i.e., there is an interval I_4 with $I_4.first = \widetilde{s}_n$ and $I_4 \models \mathtt{recover}(;\widehat{\underline{x}})$, and the final state is given by $s_{n+1} = I_4.last$.

One crucial insight into Def. 4.20 is that the re-execution yields an alternative run of the system according to Remark 4.21.

Remark 4.21. The semantics of the reset transition give an *alternative run* of the component. Assume that the i-th reset transitions yields the intervals I_0^i and I_2^i and histories h_0^i and h_2^i in step (2.) of Def. 4.20. Then the interval

$$I' = I_0^0 \fatsemi I_2^0 \fatsemi \left(I_2^0.last, I_0^1.first\right) \fatsemi I_0^1 \fatsemi I_2^1 \fatsemi \left(I_2^1.last, I_0^2.first\right) \fatsemi \cdots$$

with history

$$h' = h_0^0 \cdot h_2^0 \cdot \langle reset \rangle \cdot h_0^1 \cdot h_2^1 \cdot \langle reset \rangle \cdots$$

is an alternative run of the component that could also have happened. For each individual fragment in between two crashes Fig. 4.8 shows the corresponding fragment of the run. Note that $I_2.last \xrightarrow{\text{Reset}} s_{n+1}$ holds for the states in the figure, because the reset transition can always retract no state at all and re-execute no operation.

Several other aspects of Def. 4.20 are pointed out next.

Re-execution is optional and only permitted when at least one operation is pending. The state s_{n+1} will be synchronized according to the definition of weakly admissible components (Def. 4.3 on page 28), implying that another crash does not go back further in the history. Therefore, the split in individual segments in between crashes in Def. 4.19 is valid.

State $I_2.last$ must fall into the domain of the crash predicate. This corresponds to the intuition that a power cut can be *observed in* or needs to be *considered in* states in the domain of the crash predicate only. Expressing the crash predicate on a selected subset of states is easier for the given component and its clients as we have motivated with Sec. 4.2 and come back to in Ch. 6 and Ch. 12.

The definition of the reset transition implies the existence of a different run without a retraction, that ends in the same state s_{n+1} as visualized by Fig. 4.8.

Def. 4.20 implicitly assumes that the recovery operation of a composed component calls the recovery operations of its subcomponents only and no other operations, since other calls are missing from the history. This is not an essential limitation, it just saves notation.

A component where all states are synchronized ($\widehat{sync} \equiv true$) neither retracts nor re-executes operations. This view is used for the lowest level of specification, where the distinction between volatile and persistent memory is explicit, and the effect of a power cut is expressed as just forgetting data in volatile memory.

Def. 4.22 defines the observable behavior of a crash-aware component, which removes the operation invocations and return events of subcomponents. We denote with $h \lfloor C$ the sequence of events on component C including reset events

Definition 4.22 (Observable Behavior)**.** The observable behavior $Obs(C)$ of the component C consists of the restriction $h \lfloor C$ to the events of C of a history h with $(I, h) \in runs(C)$ for some interval I.

Note that the observable behavior of a specification component still contains concurrent operation executions, because the procedure call introduces an additional step before and after the body. Nontermination or infinite blocking is visible in the observable behavior, because only then a matching return event is missing for an invocation event, assuming that no (interrupting) reset event follows afterwards and assuming the scheduler is fair.

4.4 Correctness of Components

Linearizability and the extension crash linearizability is expressed in terms of an easily understandable, sequential specification. We use specification components according to Def. 4.6 on page 28 for this purpose and make sure that only *compatible* components are compared.

Definition 4.23 (Compatible Components)**.** Two components C and A are compatible if and only if the index set $\mathfrak{I}$ of interface operations, their input and output parameters, and the index set $\mathfrak{K}$ of internal operations are the same.[2]

The concurrent and crash behavior of a specification component is easily understood in terms of its sequential behavior according to Lem. 4.24. Intuitively, each operation takes effect at a specific point in the interval, called the *linearization point*. The other two steps of the procedure call are just stuttering steps and may be re-ordered arbitrarily.

Lemma 4.24 (Specification Components & Crash Linearization)**.** *Given a history h with $h \in Obs(A)$ of a specification component A, the history has the form*

$$h = h_0 \cdot \langle reset \rangle \cdot h_1 \cdot \ldots \cdot \langle reset \rangle \cdot h_n$$

where h_j is crash-free for all $0 \leq j \leq n$. Then there exist linearizations h_j'' of h_j and the combined history

$$h'' = h_0'' \cdot \langle reset \rangle \cdot h_1'' \cdot \ldots \cdot \langle reset \rangle \cdot h_n''$$

satisfies $h'' \in Obs(A)$. We call h'' the crash linearization *of h.*

Proof. Given a history h_j and the corresponding interval I_j, we add return events for all pending invocations if and only if the atomic body is already executed in I_j, resulting in a history h'. The history $completed(h')$ is equivalent to the sequential history h_j'' where all events of $completed(h')$ are ordered according to the point in time in I_j when the atomic block of the operation is executed. This only reorders concurrent operations and therefore respects the real-time order of $completed(h')$ and therefore h_j'' is a linearization of h_j.

The combination of all so constructed histories into history h'' as by the lemma, is also part of the observable behavior, because for each fragment I_j of the original execution, there is an interval I_j'' with $\#\, I_j = \#\, I_j''$, $I_j.first = I_j''.first$, $I_j.last = I_j''.last$ constructed by moving the invocation and return steps before and after the procedure body right before and after the step of the procedure body, respectively. Note that this re-arranging of these steps does not change the crash behavior, i.e., the same states can be targeted by the retraction transition, although at a different index in the interval, and re-execution can reach the exact same state $I_{j+1}.first$ for the crash in interval I_j. The combined interval I'' and combined history h'' then satisfy $(I'', h'') \in Obs(A)$. □

Lem. 4.24 essentially states that for each concurrent execution with crashes there is a sequential execution with the same observable behavior.

For standard linearizability proofs of concurrent objects [62] the linearization point is identified, it is proven that all program steps before and after the linearization point have no externally visible effect on the object and the linearization point itself corresponds to the entire externally visible effect of the operation.

Def. 4.25 gives a correctness criterion for a concurrent, crash-aware component C based on the relation of its observable behavior to a specification component A. This definition facilitates incremental refinement and abstraction of components as discussed in Sec. 4.5.

Definition 4.25 (Crash Linearizability & Total Correctness)**.** A component C is *crash linearizable* with respect to a compatible specification component A if and only if for each $h \in Obs(C)$ there is a crash linearization h' with $h' \in Obs(A)$. If $Obs(C) \subseteq Obs(A)$ holds, then C is *totally correct* with respect to A.

The general idea is that although the component C may have a complex concurrent implementation with many intermediate steps where crashes may occur, it can be reasoned

[2] In practice, we allow that A has less internal operations than C. All omitted internal operations have to refine the program **skip** and are therefore not observable by clients.

about in purely sequential terms. A client of C only needs to consider the runs of the component C that are sequential, because for any concurrent run the combination of Def. 4.25 and Lem. 4.24 guarantees a sequential one with the same behavior.

Note that *crash linearizability* itself is a safety property only, because the abstract system may always choose a completion of a nonterminating operation of C. Standard linearizability is also defined as a safety property only and liveness is defined separately, for example as lock-freedom or deadlock-freedom. Total correctness additionally ensures that the system does not block globally, i.e., is *deadlock free*, and has no infinite computations. The latter property is called *divergence freedom* in [12]. In the context of this thesis and lock-based algorithms under weak-fair scheduling, total correctness is used as the criterion for the correctness of a component, instead of crash linearizability.

4.5 Refinement, Compositionality & Substitution

This section first defines refinement of components and then discusses substitution of implementation components C for their specification components A in a composed system $M \multimap\!\!\circledcirc\!\!\multimap A$ with potentially additional subcomponents. A compositionality result is achieved under certain conditions.

Refinement is defined based on preserving observable behavior and therefore total correctness.

Definition 4.26 (Refinement)**.** A component C refines a compatible component A, written $A \sqsubseteq C$, if and only if the observable behavior of the component C is a subset of the observable behavior of the component A, i.e., $Obs(C) \subseteq Obs(A)$ holds.

Refinement is a preorder on components, i.e., refinement is reflexive and transitive.

A weaker version of refinement that only preserves crash linearizability is possible, too.[3] In the context of this thesis, however, the focus is on sequential components in the first part and on lock-based synchronization under weak-fair scheduling in the second part and there total correctness is more appropriate.

In Ch. 5 and Ch. 13 we will *incrementally* refine a component A towards an implementation C or, equivalently, abstract an implementation C towards the desired specification A in several steps. We will construct several intermediate components $C_1, \ldots, C_n$ and $A_1, \ldots, A_m$ with the properties (1.) to (3.).

1. $C_n \sqsubseteq \cdots \sqsubseteq C_1 \sqsubseteq C$, (Atomicity Refinement)
2. $A \sqsubseteq A_m \sqsubseteq \cdots \sqsubseteq A_1$, and (Crash Refinement)
3. $A_1 \sqsubseteq C_n$ (Data Refinement / Linearizability)

Transitivity of refinement then guarantees that the original implementation C refines the final specification A, i.e., that $A \sqsubseteq C$ holds, and therefore implies correctness of the approach. In step (1.) the atomicity of the component with respect to power failures and concurrent threads is increased by gradually collapsing several program statements into atomic blocks and is termed *atomicity refinement*. Step (3.) changes the data representation and therefore constitutes a standard *data refinement* (for sequential components) or a standard proof of *linearizability* (for concurrent components). The specification of the crash behavior is modified in step (2.) by reducing the set of synchronized states from all states in component A_0 to a more desirable subset in component A. We will usually only need one step for this *crash refinement*, i.e., m is one.

[3] Note that then an unfair interleaving should be chosen for the component semantics, because a weak-fair interleaving forces the sequential specification to complete its pending operations for every thread. Technically, the difference between weak-fair and unfair scheduling occurs in the proof of Thm. 1 below, where for a nonterminating run of a component the abstract specification cannot stall the completion of the operation infinitely long. Under unfair scheduling, however, this is not an issue.

component M
  subcomponent A_1, A_2
interface operations
  $\texttt{inc}_1(;n)$ { $A_1.\texttt{inc}(;n);$ }
  $\texttt{inc}_2(;n)$ { $A_2.\texttt{inc}(;n);$ }
recovery `recover()` { $A_1.\texttt{recover}();A_2.\texttt{recover}()$ }

component A_1
state $cnt_1, pcnt_1 : \mathbb{N}$
interface operations
  $\texttt{inc}(;n)$
    $n := cnt_1;$
    $cnt_1 :+= 1;$
    if cnt_1 even **then**
      $pcnt_1 := cnt_1;$
synchronized states cnt_1 even
crash $pcnt_1$ even $\wedge\ pcnt_1' = pcnt_1$
recovery `recover()` { $cnt_1 := pcnt_1$ }

component C_1
state $cnt_1, pcnt_1 : \mathbb{N}$
interface operations
  $\texttt{inc}(;n)$
    $n := cnt_1;$
    $cnt_1 :+= 1;$
    if cnt_1 even **then**
      $pcnt_1 := cnt_1;$

crash $pcnt_1' = pcnt_1$
recovery `recover()` { $cnt_1 := pcnt_1$ }

Figure 4.9: Counterexample to a General Compositionality Theorem where A_2 and C_2 is a renaming of A_1 and C_1, respectively.

We denote with $M\{A \mapsto C\}$ the substitution of the subcomponent A of component M with C if C refines A and is therefore syntactically compatible. Compositionality is the property that if component C refines its specification A, then $M\{A \mapsto C\}$ refines M for any component M. Or more informally, the notion that substitution of specifications by their implementations in any context preserves the behavior of the context.

$$A \sqsubseteq C \implies M \sqsubseteq M\{A \mapsto C\} \qquad \text{(Compositionality)}$$

A general compositionality result, however, is impossible.

This can be demonstrated by looking at a sequential component M with two retracting subcomponents A_1 and A_2. During a power failure the implementation C_1 of A_1 and the implementation C_2 of A_2 may choose an arbitrary prior state of their execution and re-execute several pending operations with a different outcome. There is no guarantee that C_1 and C_2 chose consistent states and that the individual re-executions yield a composed execution of C that is consistent with C's program order.

Example 4.27 (Counterexample to General Compositionality). Fig. 4.9 shows a simple counterexample for a general compositionality result using sequential components only. The component M uses two counters A_1 and A_2, each persisting only even values of the counter. The operations of component M just expose the increment operations of both components individually.

The semantics of the implementation C_1 and specification A_1 are identical, although A_1 expresses a crash as a retraction and a partial crash predicate whereas C_1 expresses it with a total crash predicate. To see this note that A_1 is forced to retract increments starting in an even value and re-execution of them is impossible, because no state in the domain of the crash predicate is reached.

The system $M\{A_1, A_2 \mapsto C_1, C_2\}$ has the run (example-run) where the first and second element of the pair denotes the counter cnt_1 of A_1 and cnt_2 of A_2, respectively.

$$(0,0) \xrightarrow{\texttt{inc}_1(0)} (1,0) \xrightarrow{\texttt{inc}_2(0)} (1,1) \xrightarrow{\texttt{inc}_2(1)} (1,2) \xrightarrow{\text{Reset}} (0,2) \qquad \text{(example-run)}$$

The synchronized predicate of $M\{A_1, A_2 \mapsto C_1, C_2\}$ is equivalent to *true*. Therefore no

retractions and re-executions are possible and the state $(0, 2)$ is the only state reachable by a reset transition from state $(1, 2)$.

The system M now has to be able to construct the same state with a reset transition starting in state $(1, 2)$. Only the state $(0, 0)$ is synchronized for the component M and therefore all states before $(1, 2)$ could be targeted by a retraction. However, only a retraction to the initial state $(0, 0)$ can yield a value of 0 for cnt_1. This precludes re-execution of the two $\texttt{inc}_2$ operations that are necessary to reach a value of 2 for cnt_2. Thus, we conclude that the transition $(1, 2) \xrightarrow{\text{Reset}} (0, 2)$ is not feasible in the component M and $M \not\sqsubseteq M\{A_1, A_2 \mapsto C_1, C_2\}$ holds.

Ex. 4.27 shows that two synchronized predicates that are not equivalent to *true* in a composition of components M poses a problem for compositionality. However, an retracting subcomponent of M and another subcomponent with a crash predicate that is not equivalent to *true* is another problem as shown by Ex. 4.28.

Example 4.28 (Counterexample to General Compositionality). We reuse the components from the previous example and Fig. 4.9. Consider the component $M' \equiv M\{A_2 \mapsto C_2\}$ where C_2 is already substituted for its implementation C_2. Component M' has one non-retracting and one retracting subcomponent. The run (example-run) given for Ex. 4.27 is also a run of M'. If we then substitute the implementation C_1 for the retracting subcomponent A_1 in M' we also get the component $M'\{A_1 \mapsto C_1\} \equiv M\{A_1, A_2 \mapsto C_1, C_2\}$ as in Ex. 4.27 and this component does not have the run. We have therefore shown that $M' \not\sqsubseteq M'\{A_1 \mapsto C_1\}$ for an example component M' with only one retracting subcomponent.

The insight provided by Ex. 4.27 and 4.28 is that a retracting subcomponent coerces its context into a specific behavior during a power failure. Therefore, the context may not further limit the reset transition. This should not come as a surprise, because the point of synchronized states is to have an *implicit* mechanism for the propagation of the crash behavior.

Def. 4.29 details restrictions on the context that admit compositionality.

Definition 4.29 (Strong Admissibility)**.** A component M is *strongly admissible* if

1. it is weakly admissible according to Def. 4.3 on page 28,
2. no subcomponent call is performed inside an atomic block,
3. only one leaf component A of C may be a retracting components according to Def. 4.6 on page 28,
4. if there is a retracting leaf component A, then all other components must have a crash predicate equivalent to *true*, and
5. if all components are non-retracting, then the crash predicate of all components must be total.

Restriction (2.) of Def. 4.29 ensures that the client component M does not presume a stricter order on the linearization points by for example assuming that two calls to the same subcomponent can be performed atomically without any interruption.

An intuitive way of stating restriction (4.) of Def. 4.29 is that if one subcomponent is retracting, the state variables of all other components are in volatile main memory. This applies to a file system where (an abstraction of) the storage device is at each step part of only one subcomponent as shown in Ch. 6 for the Flashix file system.

Restriction (5.) of Def. 4.29 ensures that one component does not interfere with the crash behavior of the other components by for example choosing a crash predicate equivalent to *false*. Technically, this is not required for a compositionality result, however, it is undesirable from a modeling perspective if one component can preclude certain crash behaviors of another component.

Refinement is compatible with this restricted form of hierarchical composition, i.e., total correctness of a component is preserved by the substitution of its subcomponents according to Thm. 1.

Theorem 1 (Compositionality)**.** *Given a non-retracting component C that is totally correct with respect to a specification component A, i.e., $A \sqsubseteq C$ is satisfied, and a strongly admissible component M with subcomponent A, then $M \sqsubseteq M\{A \mapsto C\}$ holds.*

The additional condition that the component C is non-retracting, i.e., retractions or re-executions are not used to explain the crash behavior of C, ensures that only a true implementation component is substituted. Limiting the set of synchronized states is essentially only allowed for the purpose of specification. Thm. 1 is applicable in practice, because we can substitute implementation components bottom-up.

We prove Thm. 1 for a system M with only one subcomponent A. The extension to multiple subcomponents follows from restriction (4.) and (5.) of Def. 4.29, i.e., the restrictions guarantee the existence of corresponding reset transitions for additional subcomponents.

Notation. We write $ms \oplus as$ for the combined state of the component M with exactly one subcomponent A, where ms is the state of M and as the state of A. Similarly, we write $I^M \oplus I^A$ for the composition of intervals of the individual components.

The proof of Thm. 1 is first performed for the case that A is a non-retracting component.

Proof of Thm. 1 (for $\texttt{sync}^A \equiv true$). Given the run $(I^M \oplus I^C, h) \in runs(M\{A \mapsto C\})$ then $(I^C, h\lfloor C)$ is a run of C with additional stuttering steps of M.[4] For the interval I_0^C without these additional stuttering steps $(I_0^C, h\lfloor C) \in runs(C)$ holds. The refinement $A \sqsubseteq C$ guarantees a run $(I_0^A, h\lfloor C) \in runs(A)$. The interval I_0^A can be padded with the additional stuttering steps of M and of C into I^A. The general idea is to align the steps for invoke events, linearization points and return events in I^A with their corresponding steps in I_0^C and add stuttering steps in between. If an operation of C does not terminate so does the corresponding call of A and therefore no padding is necessary. Note that every terminating call of A has exactly three system steps per operation while the corresponding call of C has at least three steps. The reset steps are then automatically aligned. Then $(I^M \oplus I^A, h)$ is a run of M with additional stuttering steps. Removing them yields a run of M. □

We now shift our attention to a retracting subcomponent A.

Proof of Thm. 1 (for $\texttt{sync}^A \not\equiv true$). The general approach is the same, i.e., we extract the run of C from a run of $M\{A \mapsto C\}$, find a matching run for A and reintegrate it yielding a run of $M\{A \mapsto C\}$ with the same observable behavior. We focus here on the reset transitions.

Given a reset transition $ms \oplus cs \xrightarrow{\text{Reset}} ms' \oplus cs'$ of $M\{A \mapsto C\}$, then there is also a reset transition $cs \xrightarrow{\text{Reset}} cs'$ of C, since $M\{A \mapsto C\}$ does not have any retraction or re-executions.[4] By refinement $A \sqsubseteq C$ there is also a matching reset transition $as \xrightarrow{\text{Reset}} as'$ of the run of A. We now have to show that $ms \oplus as \xrightarrow{\text{Reset}} ms' \oplus as'$ also holds, i.e., that it is possible to construct intermediate states in accordance with Fig. 4.8 on page 36 for component M based on the corresponding intermediates states of A. The idea is that M retracts the same transitions that A does. Afterwards the pending operations of A are re-executed according to the reset transition of A and then C stops re-execution. The subsequent crash transition can restore the same state that $M\{A \mapsto C\}$ does, because $crash^M \equiv true$ and recovery restores the state $ms' \oplus as'$. So the trick is basically that the state reached by M before the crash transition is essentially irrelevant due to the specific crash predicate of M. □

Note that Thm. 1 also holds for the weaker version of refinement that only preserves crash linearizability.

[4] Note that this extraction assumes that the recovery operation of $M\{A \mapsto C\}$ calls the recovery operation of C only and no additional operations. As stated above, this limitation can be removed if the intermediate states of the recovery operation and their history is added to runs.

4.6 Invariants, Preconditions & Assertions

In this section we will ensure that every component M calls its subcomponents within their precondition and that all other assertions are satisfied when the execution reaches them. Additionally, invariants that hold either in every step of the execution or in between steps of a sequential component may be established in this step.

In the following we write

component C
invariant $inv(\underline{\widehat{x}})$

for a sequential invariant inv of a sequential or specification component C in the variables of C and all its subcomponents. We implicitly assume that the invariant inv contains the invariants given for all subcomponents of C.

Violations of preconditions and assertions are represented as nontermination in the operations of a component. The proof obligations of Lem. 4.30 therefore ensure that all operations terminate, too.

Lemma 4.30 (Invariants, Assertions & Preconditions for Sequential Components)**.** *For sequential components with sequential invariant $inv(\underline{\widehat{x}})$ the proof obligations*

1. $\langle\!|\, \texttt{init}^C(;\underline{\widehat{x}}) \,|\!\rangle \left(inv(\underline{\widehat{x}}) \wedge \widehat{sync}(\underline{\widehat{x}})\right)$
2. $pre_i(\underline{in}, \underline{\widehat{x}}) \wedge inv(\underline{\widehat{x}}) \rightarrow \langle\!|\, \texttt{Op}_i^C(\underline{in}; \underline{\widehat{x}}, \underline{out}) \,|\!\rangle\, inv(\underline{\widehat{x}})$
3. $guard_k(\underline{\widehat{x}}) \wedge inv(\underline{\widehat{x}}) \rightarrow \langle\!|\, \widehat{\texttt{IOp}_k}^C(;\underline{\widehat{x}}) \,|\!\rangle\, inv(\underline{\widehat{x}})$
4. $inv(\underline{\widehat{x}}_0) \wedge \widehat{crash}(\underline{\widehat{x}}_0, \underline{\widehat{x}}) \rightarrow \langle\!|\, \texttt{recover}^C(;\underline{\widehat{x}}) \,|\!\rangle \left(inv(\underline{\widehat{x}}) \wedge \widehat{sync}(\underline{\widehat{x}})\right)$

are shown for each interface operation $\texttt{Op}_i^C$ with precondition pre_i of the component C and for each internal operations $\widehat{\texttt{IOp}_k}^C$ with guard $guard_k(\underline{\widehat{x}})$ of C or any of its subcomponents. The proof obligations ensure termination, establish the sequential invariant $inv(\underline{\widehat{x}})$ and all assertions, including preconditions of subcomponents. □

The proof obligations of Lem. 4.30 show that the invariants of a specification subcomponent A hold in every step of a run of C, because the state of A is not modified by steps of C and Lem. 4.30 (for A) then propagates the invariant of the subcomponent over A's atomic steps.

For atomic blocks of a sequential component the proof obligations of Lem. 4.30 ensure that the atomic blocks terminate, i.e., it is possible to show that a component is actually a specification component according to Def. 4.6 on page 28.

For a concurrent component we use rely/guarantee reasoning to establish concurrent invariants. In the following we write

component C
concurrent invariant $inv(\underline{\widehat{X}})$
rely $rely(Tid, \underline{\widehat{X}}, \underline{\widehat{X}}')$

for a concurrent invariant inv and rely condition $rely$ of a concurrent component C with (flexible) state variables $\underline{X}$. The rely condition may refer to the thread identifier Tid. For the guarantee the conjunction of all rely conditions of all other threads is used.

Instead of termination as for sequential components, we first prove divergence freedom for each thread in an invariant proof.

Lemma 4.31 (Invariants, Assertions & Preconditions for Concurrent Components)**.** *For a concurrent component with concurrent invariant $inv(\underline{\widehat{X}})$ the proof obligations (1.) and (4.) of*

Lem. 4.30 are shown and

$$(2'.)\quad pre_i(\underline{In}, \widehat{\underline{X}}) \wedge inv(\widehat{\underline{X}})$$
$$\rightarrow \langle rely(Tid, \widehat{\underline{X}}', \widehat{\underline{X}}''),\ guar(Tid, \widehat{\underline{X}}, \widehat{\underline{X}}'),\ inv(\widehat{\underline{X}}),\ false,\ \mathtt{Op}_i^C(\underline{In}; \widehat{\underline{X}}, \underline{Out})\rangle\ true$$
$$(2''.)\quad pre_i(\underline{In}, \widehat{\underline{X}}_0) \wedge rely(Tid, \widehat{\underline{X}}_0, \widehat{\underline{X}}_1) \rightarrow pre_i(\underline{In}, \widehat{\underline{X}}_1)$$
$$(3'.)\quad guard_i(\widehat{\underline{X}}) \wedge inv(\widehat{\underline{X}})$$
$$\rightarrow \langle rely(Tid, \widehat{\underline{X}}', \widehat{\underline{X}}''),\ guar(Tid, \widehat{\underline{X}}, \widehat{\underline{X}}'),\ inv(\widehat{\underline{X}}),\ false,\ \mathtt{IOp}_k^C(; \widehat{\underline{X}})\rangle\ true$$
$$(5.)\quad rely(tid, \underline{x}, \underline{x})$$
$$(6.)\quad rely(tid, \underline{x}_0, \underline{x}_1) \wedge rely(tid, \underline{x}_1, \underline{x}_2) \rightarrow rely(tid, \underline{x}_0, \underline{x}_2)$$

where

$$guar(Tid, \widehat{\underline{X}}, \widehat{\underline{X}}') \equiv \forall Tid_0 \neq Tid.\ rely(Tid_0, \widehat{\underline{X}}, \widehat{\underline{X}}')$$

for the respective operations. The proof obligations ensure divergence freedom, establish the concurrent invariant $inv(\widehat{\underline{X}})$ and all assertions, including preconditions of subcomponents, and show that atomic sections terminate if the guard is satisfied. Furthermore, it guarantees that all preconditions are stable over steps of other threads. □

Note that the runs predicate $runs \equiv false$ allows each operation to block under any condition, but ensures that no infinite computations occur otherwise. Ch. 13 will show that most of the rely condition can be automatically derived by ownership annotations for a component.

4.7 Related Work

Crashes & Recovery Koskinen et al. [80] use model checking to guarantee that a sequential system is crash-recoverable. In the event of a power failure the program is re-executed and correctness is stated as a successful re-execution of the program. This means that a crash-recoverable program should not exhibit any new behavior on a re-execution after a power failure. It is difficult to use this criterion for a modular and compositional verification of components with externally callable interface. The purpose of the re-execution in this chapter is different, the re-execution should yield a desirable state where crashes are easy to specify and consider.

Marić and Sprenger [90] model crashes as exceptions in the context of a sequential transactional hardware manager. The exception handler is used to perform the recovery of the system. This essentially corresponds to the state-based specification approach for sequential, crash-aware components. A generalization to concurrent systems of this modeling approach seems difficult, since power failures affect all threads of execution at the same time, but exceptions are handled locally by each thread.

The specification mechanisms used for the FSCQ file system [27, 28, 26] explicitly stores a history of the system. The disk model similarly stores all possible values of a sector and during a power failure nondeterministically chooses one of the possible values. This allows for re-ordering of disk writes, which in the framework of this chapter is only possible with the state-based approach. Bornhold et. al [18] model crashes on the level of abstraction of POSIX by maintaining a list of update events. In the event of a power failure a re-ordering of a prefix of the update events is applied to the initial state (or the state after the last power failure). In Amani et al. [11, 9] the effect of the operations is captured similarly, as a list of state transformers, although power failures and recovery from them is not considered formally. Such a re-ordering of operations can currently only be specified with the state-based approach for crash-aware components. In future work, we intend to extend the operations-based approach to also allow for the re-ordering of operations. However, a compositionality result is more difficult to achieve in this case.

In several other approaches [124, 101] the volatile and durable state is explicitly disjoint in the logic, and therefore also in the models. This impedes a refinement-based approach, because it limits the ability for data refinement, i.e., the volatile and durable data structures must always be refined separately. The modeling approach presented in this chapter does not enforce a distinction between volatile and durable state.

Linearizability, Serializability & Persistency Several other correctness criteria for concurrent objects with persistent data and crashes have been proposed. In the context of process-local crashes there is *recoverable linearizability* [16] and *strict linearizability* [8] and for message passing systems *persistent and transient atomicity* [57]. Each of these criteria assumes that crashes are local to a component. This matches poorly to file systems, where a power failure of the entire system and the correct recovery from such a failure is at the core of the file system's correctness. The first to consider linearizability in the context of persistent objects with crashes of the full system is Israelevitz et al. [69]. They consider the emerging technology of byte-addressable Non-volatile Random Access Memory (= NVRAM), which allows for the implementation of highly concurrent algorithms on this new form of persistent memory. They define *durable linearizability* and *buffered durable linearizability*. Both criteria do not allow for additional effects as part of a system crash, since they require that the history with crash events removed is linearizable, i.e., is a potential execution of the system. This restriction does not apply to the components of the Flashix file system as explained in more detail in subsequent chapters. There crashes may have far reaching consequences. This thesis contributes a specification mechanism with synchronized states and the crash predicate for state-based components, which simplifies the application to the Flashix case study. Note that the criterion of *buffered durable linearizability* does allow for retractions of operations. However, there is no limit to the amount of operations that may be retracted. This is also not applicable to a file system, where it is necessary to show that all operations are persisted if the user requests synchronization with the POSIX operation *sync* or *fsync*. The criterion of durable linearizability has the property that it is local, i.e., a general substitution theorem holds. For buffered durable linearizability this does not hold. Thm. 1 shows that at least for a reasonable subset of composed components with only one persistent storage substitution is possible.

Serializability and strict serializability [106] are correctness criteria for transactional, concurrent systems. Strict serializability corresponds to linearizability if a transaction is reinterpreted as an individual operation of a component.

Internal & External Operations Internal operations are similar to the events of Event-B [7], which are also triggered internally when a guard is satisfied. The classical B-Method [6] has internal operations as well as externally callable operations. Externally callable operations are also standard in formalism based on data refinement [64, 60], such as Z [133, 40, 41].

Part I

Verification of Crash-Safety & Caching in File Systems

5 Refinement of Sequential, Crash-Aware Components

Summary. This chapter discusses three types of refinements for sequential, crash-aware components. First, criteria for the refinement of the atomicity of operations with respect to power failures are introduced. Afterwards, data refinement with crashes is discussed. Finally, refinement of the expression of the reset transition, i.e., the switch from a state-based to an operations-based view of a crash, is explained. Taken together, it is possible to refine a sequential specification component with an operations-based view on the crash to an actual implementation with potential power failures in any intermediate state.

Publications. This Chapter is based on the publications [109, 45, 47, 107]. It extends [107] to facilitate the applicability to concurrent components. The hierarchy of criteria for increasing the atomicity of a component with respect to power failures, also generalizes our previous work [47, 107].

Contents

5.1 Overview

In general, a refinement step can change the atomicity of the operations of the component, the data representation as well as change the view of a crash, since only the observable behavior must be preserved. The generality of having all three changes in abstraction is only needed for a uniform definition of refinement.

In practice, Fig. 5.1 depicts our approach to the refinement of a large, crash-aware system. We will first increase the atomicity of a component C to C', then abstract its data representation to a component A', which

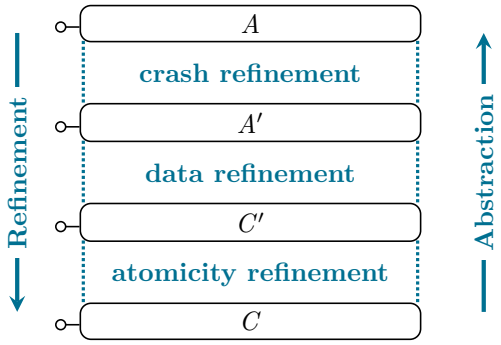

Figure 5.1: Refinement Approach for Large, Crash-Aware Systems

then allows us to abstract the specification of a crash to the final specification A. For each refinement step invariants and assertions of the component may be used to facilitate and structure the proof of refinement.

The rest of this section is structured as follows. Sec. 5.2 considers *atomicity refinement*, then Sec. 5.3 gives proof obligations for *data refinement* that is aware of crashes. The section concludes with Sec. 5.4, which defines *crash refinement*.

5.2 Atomicity Refinement

This section introduces several criteria that allow us to increase the atomicity of a component without losing behavior in the event of a power cut. These criteria work well in the context of concurrent components as discussed in Ch. 13. A calculus for proving these criteria for a component is given.

In general if we replace two consecutive single-step programs $p; q$ by one atomic block **atomic** $\{ p; q \}$, then we might lose the states reachable by a crash from the intermediate state between p and q. For example in Fig. 5.2 (solid arrows only) the state s_1' reached from s_1 by a crash and subsequent recovery might be lost. In order to ensure that no such crash behavior is omitted when the atomicity of a component is increased, we make sure that for each removed intermediate s_1 another state exists that *subsumes* the crash behavior of s_1. Such a state is usually in the vicinity of s_1, i.e., it is either s_0 or s_2 in the figure, if p or q do not persist any data and most operations do not.

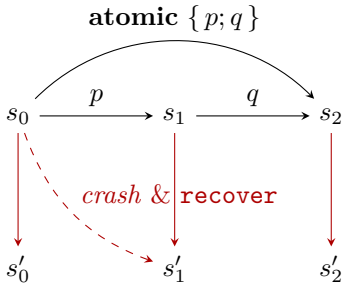

Figure 5.2: Atomicity & Crashes in Intermediate States

This section is structured as follows: First, several concepts are defined that relate the crash behavior of two states. Afterwards, the criterion of R-atomicity, which gives rise to *atomicity refinement*, is discussed. Then the three stronger criteria neutrality, retractability and introducibility are defined and their crucial properties are proven. A calculus for sequential programs is introduced that uses the invariants and assertions of a component.

5.2.1 $\mathcal{R}$-Subsuming and $\mathcal{R}$-Equivalent States

In the following $\mathcal{R} \subseteq S \times S$ is a binary relation over states. The idea is essentially that the crash and subsequent recovery of a component will be substituted for $\mathcal{R}$, or more precisely we assume that $\mathcal{R}$ is given by a formula R chosen according to Formula (R-is-*crash*-`recover`).

$$R(\underline{x}, \underline{x}') \leftrightarrow \exists \widetilde{\underline{x}}.\ \mathit{crash}(\underline{x}, \widetilde{\underline{x}}) \wedge \langle \texttt{recover}(; \widetilde{\underline{x}}) \rangle\, \underline{x}' = \widetilde{\underline{x}} \qquad (R\text{-is-}\mathit{crash}\text{-}\texttt{recover})$$

The formula states that the state $\underline{x}'$ is reached via the crash and subsequent recovery by an intermediate state $\widetilde{\underline{x}}$. The relation over states $\mathcal{R}$ is defined by Equation ($\mathcal{R}$).

$$\mathcal{R} = \{(s, s') \mid (s, s') \models R(\underline{x}, \underline{x}')\} \qquad (\mathcal{R})$$

The definitions, however, are applicable to arbitrary relations. If a term that refers to R, such as $\mathcal{R}$-reachable, is applied to the concrete instance for crash and subsequent recovery given by Eqn. (R-is-*crash*-`recover`), then the term "crash-recover" is used as a shorthand, as e.g. in crash-recover-reachable.

In general we assume that R is extended to additional variables $\underline{y}$ with $\underline{y} \cap \underline{x} = \emptyset$ by allowing arbitrary changes to $\underline{y}$.

Definition 5.1 ($\mathcal{R}$-Reachability)**.** A state s' is $\mathcal{R}$-reachable from a state s if and only if $s \xrightarrow{\mathcal{R}} s'$ holds. The set of $\mathcal{R}$-reachable states of s is $\mathcal{R}(s)$.

Definition 5.2 ($\mathcal{R}$-Subsumption)**.** A state s $\mathcal{R}$-subsumes a state s', written $s' \sqsubseteq_{\mathcal{R}} s$, if and only if the $\mathcal{R}$-reachable states of s' are a subset of those of s, i.e., $\mathcal{R}(s') \subseteq \mathcal{R}(s)$ holds.

If for example the state s_0 subsumes s_1, then the dotted arrow in Fig. 5.2 from s_0 to s_1' exists. Analogously if s_2 subsumes s_1. Fig. 5.3 shows the set of $\mathcal{R}$-reachable states of s (black hatched area) and s' (blue hatched area) if s $\mathcal{R}$-subsumes s'.

The relation $\sqsubseteq_{\mathcal{R}}$ is a preorder, i.e., it is reflexive and transitive.

One property of $\sqsubseteq_{\mathcal{R}}$ is that it is possible to split the relation into a composition $\mathcal{R}_1 \fatsemi \mathcal{R}_2$ of two relations $\mathcal{R}_1$ and $\mathcal{R}_2$ and only show $s' \sqsubseteq_{\mathcal{R}_1} s$ if possible, in order to infer $s' \sqsubseteq_{\mathcal{R}_1 \fatsemi \mathcal{R}_2} s$.

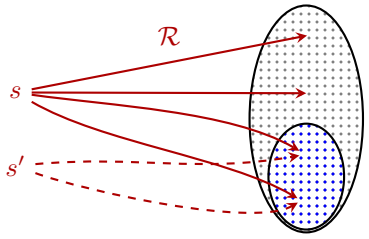

Figure 5.3: $\mathcal{R}$-Reachable States of s and s' with $s' \sqsubseteq_{\mathcal{R}} s$

Lemma 5.3 ($\mathcal{R}_1 \fatsemi \mathcal{R}_2$-Subsumption)**.** $\sqsubseteq_{\mathcal{R}_1} \subseteq \sqsubseteq_{\mathcal{R}_1 \fatsemi \mathcal{R}_2}$

Proof. Given states s and s' where s $\mathcal{R}_1$-subsumes s', i.e., $\mathcal{R}_1(s') \subseteq \mathcal{R}_1(s)$ holds, then by

$$(\mathcal{R}_1 \fatsemi \mathcal{R}_2)(s') = \mathcal{R}_2(\mathcal{R}_1(s')) \subseteq \mathcal{R}_2(\mathcal{R}_1(s)) = (\mathcal{R}_1 \fatsemi \mathcal{R}_2)(s)$$

it follows that s also $(\mathcal{R}_1 \fatsemi \mathcal{R}_2)$-subsumes s'. □

Lem. 5.3 allows us to prove crash-recover-subsumption by only considering crash-subsumption, which is far easier, because the recovery is usually given by a rather complex program and verifying that a specific change in the state before recovery does not alter the result is rather difficult. The formula *crash* is usually far simpler.

Definition 5.4 ($\mathcal{R}$-Equivalence)**.** Two states s and s' are $\mathcal{R}$-equivalent, written $s \equiv_{\mathcal{R}} s'$, if and only if $\mathcal{R}(s) = \mathcal{R}(s')$ holds, i.e., the sets of $\mathcal{R}$-reachable states are the same.

Note that $\equiv_{\mathcal{R}}$ is an equivalence relation.

For two $\mathcal{R}$-equivalent states either one may be removed by increasing the atomicity of the program without altering the behavior on a power cut.

5.2.2 *R*-Atomicity & Refinement

The next step is to systematically find a subsuming state to an intermediate state s by looking at the programs leading to or following the state and then remove s just as done in Fig. 5.2 on page 50 for state s_1. The most general case whereby removing the intermediate states does not alter the crash behavior at all is given by Def. 5.5.

Definition 5.5 (R-Atomic)**.** A program p is R-atomic if and only if for each intermediate state s_i of a finite execution $I = (s_0, \ldots, s_n)$ either

- $s_i \sqsubseteq_{\mathcal{R}} s_0$ holds, or
- there is a (potentially different) finite execution $I' = (s_0', \ldots, s_m')$ starting in $s_0' = s_0$ with $s_i \sqsubseteq_{\mathcal{R}} s_m'$.

We denote with

$$\varphi, inv \vdash p : \curvearrowright_R$$

the judgment that program p is R-atomic in the context where the assertion φ holds initially and the invariant *inv* always holds.

The symbol $\curvearrowright\!\!\!\!\curvearrowright$ is chosen to indicate that crash behavior of intermediate states is either subsumed by the initial state of the execution or possible other final states of other completions.

For the calculus it is possible to transfer some knowledge such as the assertions φ that hold in the initial state of p and the invariants *inv* that always hold for the entire component that p is part of.

Lem. 5.6 states that a crash-recover-atomic program p, as the name implies, can be replaced by an atomic block **atomic** $\{\, p \,\}$ without loosing behavior on a power cut.

Lemma 5.6 (R-Atomic & Atomicity)**.** *If a program p is R-atomic, then*

$$[\![p]\!]^{\lightning}_{\lightning}\mathcal{R} \subseteq [\![\textbf{\textit{atomic}}\ \{\, p \,\}]\!]^{\lightning}_{\lightning}\mathcal{R}$$

holds.

Proof. Given a state s_i of a finite execution $(s_0, \ldots, s_n)$ of p then there is another execution $(s'_0, \ldots, s'_m)$ with $s'_0 = s_0$ and either $s_i \sqsubseteq_{\mathcal{R}} s_0$ or $s_i \sqsubseteq_{\mathcal{R}} s'_m$ is satisfied. The $\mathcal{R}$-reachable states of s_i are a subset of those of s_0 or s'_m and (s_0, s'_m) is an execution of **atomic** $\{\, p \,\}$.

Every infinite execution $(s_0, \ldots)$ of p, is also an execution of **atomic** $\{\, p \,\}$ and therefore the property holds trivially. □

All programs that only take a single step, such as a single assignment, are trivially R-atomic.

Remark 5.7. Note that the reverse implication of Lem. 5.6 does not hold. The reason is that the crash behavior of an intermediate state s_i might not by subsumed by the initial state of the execution or some final state reachable from the initial state, but maybe by several of those. Consider for example the program

$$p \equiv \{\, n \;:=\; 1 \;;\; n \;:=\; 2 \,\} \qquad \text{with} \qquad \mathcal{R} = \{\, (0,0), (1,0), (1,1), (2,1), (2,2) \,\},$$

starting in a state where $n = 0$ holds. Then the potential set of intermediate values of n is $S = \{0, 1, 2\}$. The set of $\mathcal{R}$-reachable states therefore is also S. The program **atomic** $\{\, p \,\}$ does not have the intermediate state $n = 1$, but the set of $\mathcal{R}$-reachable states is still S and the subset relation of Lem. 5.6 holds. The intermediate state $n = 1$, however, is not $\mathcal{R}$-subsumed by either $n = 0$ or $n = 2$, only by the combination of both. This seems to be a rare case though.

Thm. 2 lifts the substitution of **atomic** $\{\, p \,\}$ for p to the level of entire components, where synchronized states and retractions need to be considered.

Theorem 2 (Atomicity Refinement of Sequential Components)**.** *If the program p is crash-recover-atomic in the context of a component C with invariant inv that always holds and with assertion φ before every occurrence of p, i.e.,*

$$\varphi, \mathit{inv} \vdash p : \curvearrowright\!\!\!\!\curvearrowright_R \qquad \textit{for } R \textit{ of Eqn. (R-is-crash-recover) on page 50}$$

*is satisfied, then the component C refines the component $A \coloneqq C\{p \mapsto$ **atomic** $\{\, p \,\}\}$ where **atomic** $\{\, p \,\}$ is substituted for p in C, i.e.,*

$$C\{p \mapsto \textbf{\textit{atomic}}\ \{\, p \,\}\} \sqsubseteq C$$

*holds and inv is also an invariant of $C\{p \mapsto$ **atomic** $\{\, p \,\}\}$.*[1]

Note that the component $C\{p \mapsto$ **atomic** $\{\, p \,\}\}$ is only more atomic than C if we prove termination afterwards, as is done during the invariant and assertion proofs of Sec. 4.6.

[1] If the recovery operation is not considered atomic, then it is necessary to first prove that recovery is crash-recover-atomic. This allows substituting the recovery operation with its atomic version. Another approach is to show crash-atomicity for p, then substitution in a non-atomic recovery operation is also possible.

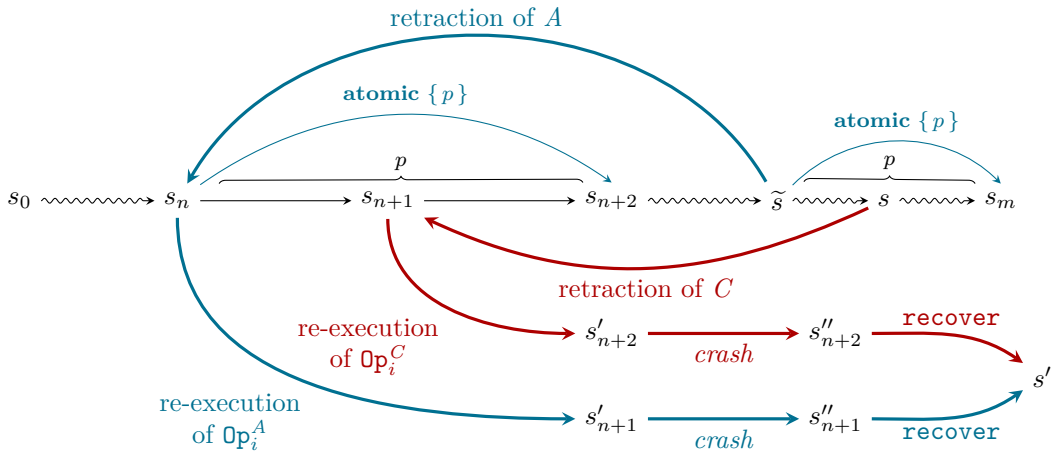

Figure 5.4: Constructing a reset transition for $A = C\{p \mapsto \textbf{atomic}\ \{\, p \,\}\}$ based on a reset transition of C: The original execution of C is depicted in as **black arrows**. The execution of A makes the **blue** detours from the run of C, if the program p is the source or target of the retraction transition of C.

Proof. The sequential input/output behavior of the component C is obviously preserved by the substitution. Furthermore, A has fewer states than C, and therefore the invariant inv has the invariant inv, too.

Assume that the reset transition $s \xrightarrow{\text{Reset}} s'$ of C leads from an intermediate state s of the execution of operation $\mathtt{Op}_i^C$ to the state s'. We construct a reset transition $\widetilde{s} \xrightarrow{\text{Reset}} s'$ of A starting in an intermediate state $\widetilde{s}$ of the corresponding execution of $\mathtt{Op}_i^C$. This shows that the crash behavior is preserved by the substitution. The interesting case of this argument is visualized in Fig. 5.4 and should be followed alongside the semantics of reset transitions shown in Fig. 4.8 on page 36.

If the intermediate state s of C is not produced during the execution of p, we choose $\widetilde{s} = s$. Otherwise, we use the state directly before the execution of p as $\widetilde{s}$. This state is also part of the corresponding execution in A. Similarly, if the target of the retraction transition of C is not part of the execution of p, we chose the same state for the retraction and the state directly before the execution of p otherwise. In the former case the remaining transitions of the reset are exactly identical. In the latter case Lem. 5.6 guarantees that the re-execution of **atomic** $\{\, p \,\}$ of A can reach the same state s' after crash and recovery as the re-execution of p of C.

Note that the executions of A have fewer synchronized states and that therefore more retractions are allowed. □

The drawback of the criterion of R-atomicity is that it cannot be used to incrementally construct ever larger atomic blocks, i.e., if programs p and q are R-atomic, it does not follow that $\{p; q\}$ is also R-atomic. Consider the example programs p and q with

$$p \equiv \quad n \coloneqq n+1 \qquad \text{and} \qquad q \equiv \quad n \coloneqq n-1 \qquad \text{and} \qquad \mathcal{R} = \{(n, n') \mid n' \leq n\}.$$

Then

$$[\![\textbf{atomic}\ \{\, p; q \,\}]\!]^{\lightning}\, {}_{9}\mathcal{R} = [\![\textbf{skip}]\!]^{\lightning}\, {}_{9}\mathcal{R} = \mathcal{R} \quad \neq \quad [\![p]\!]^{\lightning}\, {}_{9}\mathcal{R} = [\![p; q]\!]^{\lightning}\, {}_{9}\mathcal{R}$$

clearly holds, since for example the right-hand side contains the pair $(0, 1)$, while the left-hand side does not.

5.2.3 R-Neutrality, R-Retractability and R-Introducibility

In the following we will therefore look at three stronger criteria, which have the property that if p and q satisfy the criterion †, then $\{p;q\}$ also satisfies †. All criteria imply R-atomicity and therefore Lem. 5.6 is applicable.

Fig. 5.5 depicts this hierarchy and the symbols used for each criterion.

The criteria also admit a calculus that in practice can be applied mostly automatically.

Ch. 6 shows how the calculus is applied to the Flashix file system and the kind of proof obligations that remain to be shown.

The strongest criterion is R-neutrality, which basically states that the crash behavior is not changed in any of the steps of a program, and is given by Def. 5.8.

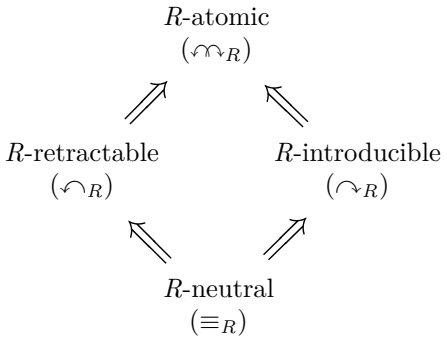

Figure 5.5: A Hierarchy of Criteria for Atomicity

Definition 5.8 (R-Neutral)**.** A program p is R-neutral if and only if for finite executions $(s_0, \ldots, s_n)$ the states $\{s_i\}_i$ are $\mathcal{R}$-equivalent. We denote with $\varphi, inv \vdash p : \equiv_R$ the judgment that p is R-neutral.

The symbol $\equiv$ indicates that the crash behavior is the same throughout an execution of the program.

Examples of (parts of) crash-recover-neutral (steps of) programs include the evaluation of tests (in e.g. **if** and **while** programs), **skip** and assignments to variables that are irrelevant to the crash behavior. Irrelevant variables are e.g. local variables and the RAM state of the component.

A crash-recover-neutral program p may always be combined into an atomic block with the preceding or subsequent program, but not necessarily with both at the same time, since the combination is not always crash-recover-neutral again.

A more general, but less intuitive criterion is given by R-retractability in Def. 5.9. If p is R-retractable, then the program $\{p;q\}$ may be replaced by **atomic** $\{\, p; q \,\}$ assuming q is R-atomic. For example q could be an assignment.

Definition 5.9 (R-Retractable Programs)**.** A program p is R-retractable if and only if for each finite execution $(s_0, \ldots, s_n)$ the initial state subsumes the crash behavior of all following states, i.e., $s_i \sqsubseteq_{\mathcal{R}} s_0$ holds for $0 \leq i \leq n$. We denote with $\varphi, inv \vdash p : \curvearrowleft_R$ the judgment that p is R-retractable.

The symbol $\curvearrowleft$ signifies that the initial state of every execution already exhibits the crash behavior of the entire execution.

For example synchronizing programs p are R-retractable, since after execution of p it is ensured that the data is persisted, i.e., the behavior on a power failure is just identity of the state, whereas before p the data is only potentially persisted.

Another criterion is R-introducibility. It is in some sense symmetrical to R-retractability and also weaker than R-neutrality. If q is R-introducible, then $\{\, p; q \,\}$ may be replaced by **atomic** $\{\, p; q \,\}$ assuming p is R-atomic.

Definition 5.10 (R-Introducible)**.** A program p is R-introducible if and only if for each finite execution $I = (s_0, \ldots, s_n)$ and for each state s_i there is a finite execution $I' = (s'_0, \ldots, s'_n)$ starting in $s'_0 = s_0$ with $s_i \sqsubseteq_{\mathcal{R}} s'_n$ holds. We call the execution I' the *R-subsuming execution of s_i*. We denote with $\varphi, inv \vdash p : \curvearrowright_R$ the judgment that p is R-introducible.

The symbol $\curvearrowright$ indicates that the crash behavior of initial and intermediate states of an execution is subsumed by final states of a (potentially other) execution of the program.

Remark 5.11. Note that Def. 5.10 facilitates the interaction between R-introducibility and atomic blocks by postponing proofs of termination. This allows us for example to infer R-introducibility of **while** φ **do** $\{\, p \,\}$ based solely on R-introducibility of p. This is also a minor change to our definition in [47, 107]. It is made possible by the semantics of atomic blocks of Def. 3.5 on page 20. Another change to the definition is that it is not necessary to complete the execution starting in s_i; it is instead sufficient to find a possibly different execution from the same initial state.

Lem. 5.12 states that the implications shown in the hierarchy in Fig. 5.5 in fact hold.

Lemma 5.12 (Hierarchy of Criteria for Atomicity)**.** *If program p is R-neutral it is also R-retractable and R-introducible. If p is either R-retractable or R-introducible it is also R-atomic.*

Proof. By definition of the respective criteria. □

Remark 5.13. Note that if a program p is R-introducible and R-retractable it is not necessarily also R-neutral. Take for example the program

$$p \equiv \quad \textbf{skip} \vee \{\, n \;:=\; n-1 \,\}$$

and the relation $R = \{(n, n') \mid n' \leq n\}$. Program p is R-retractable since decreasing n only restricts the available values of n after R. Furthermore, p is R-introducible, which is ensured by the **skip** and Lem. 5.20. However, the program is not R-neutral, since the set of values available after R *might* have been reduced.

As stated already R-atomicity does not admit sequential composition. Lem. 5.14 states, however, that all other criteria in fact propagate to the sequential composition.

Lemma 5.14 (Atomicity Criteria & Sequential Composition)**.** *If p and q are both R-neutral, both R-retractable or both R-introducible programs, then so is $\{p; q\}$.*

Proof. For R-neutrality this is obvious by its definition, since only two intervals with R-equivalent states are concatenated.

Assume program p and q are R-retractable. Every finite execution $I = (s_0, \ldots, s_n)$ can be split into $I' = (s_0, \ldots, s_j)$ and $I'' = (s_j, \ldots, s_n)$ where I' and I'' is the execution of p and q, respectively. For all s_i with $i \leq j$, $s_i \sqsubseteq_{\mathcal{R}} s_0$ by R-retractability of p. For all other s_i, the chain $s_i \sqsubseteq_{\mathcal{R}} s_j \sqsubseteq_{\mathcal{R}} s_0$ holds by R-retractability of both programs.

Assume that p and q are R-introducible. Given a finite execution $I = (s_0, \ldots, s_n)$, then there is an index j with $0 \leq j \leq n$ splitting the interval into a part where p runs and a part where q runs.

For states s_i with $i \geq j$, by R-introducibility of q there is an R-subsuming execution I' of s_i starting in s_j. The interval $(s_0, \ldots, s_j) \fatsemi I'$ is the sought-after R-subsuming execution.

If $i < j$ holds, then there is an R-subsuming execution I' of s_i starting in s_0. If the execution $I' = (s'_0, \ldots, s'_m)$ is finite with $s_i \sqsubseteq_{\mathcal{R}} s'_m$, then s'_m has an R-subsuming interval I'' starting in s_m, since q is also R-introducible. $I' \fatsemi I''$ is then the sought-after R-subsuming execution. □

If one of the programs in $\{\, p; q \,\}$ is R-atomic only the weaker form of sequential composition of Lem. 5.15 holds.

Lemma 5.15 (R-Atomicity & Sequential Composition)**.** *If p is R-atomic and q is R-introducible or p is R-retractable and q is R-atomic, then $\{p; q\}$ is R-atomic.*

Proof. The proof is similar to the proof of Lem. 5.14. □

The criteria also propagate over loops as shown by Lem. 5.16.

Lemma 5.16 (Atomicity Criteria & Loops)**.** *If p is R-neutral, R-introducible, or R-retractable, then so is* ***while*** φ ***do*** $\{\, p \,\}$ *and* p^**.*

Proof. For a finite execution I of **while** φ **do** $\{\, p \,\}$ there is a finite n with $I \models (\mathbf{skip}; p)^n ; \mathbf{skip}$ by unfolding the **while**-loop until it terminates. Then Lem. 5.14 can be applied. The proof for p^* is similar. □

Recursive calls require an inductive argument. Since nontermination is excluded from consideration by each of the criteria, Lem. 5.17 just replaces the recursive call with an assignment that captures all possible effects of the recursive call, while still satisfying the atomicity criterion in question. In order to state the lemma we define the formula $\mathrm{ACrit}^{\dagger}_{R}(\underline{x}, \underline{x}')$ for $\dagger \in \{\curvearrowright\!\!\!\curvearrowleft, \curvearrowleft, \curvearrowright, \equiv\}$ by Equations. ($\mathrm{ACrit}^{\dagger}_{R}$).

$$
\begin{aligned}
\mathrm{ACrit}^{\equiv}_{R}(\underline{x}, \underline{x}') &\equiv \forall \underline{x}''.\ R(\underline{x}, \underline{x}'') \leftrightarrow R(\underline{x}', \underline{x}'') && (\mathrm{ACrit}^{\dagger}_{R})\\
\mathrm{ACrit}^{\curvearrowright}_{R}(\underline{x}, \underline{x}') &\equiv \forall \underline{x}''.\ R(\underline{x}, \underline{x}'') \rightarrow R(\underline{x}', \underline{x}'')\\
\mathrm{ACrit}^{\curvearrowleft}_{R}(\underline{x}, \underline{x}') &\equiv \forall \underline{x}''.\ R(\underline{x}, \underline{x}'') \leftarrow R(\underline{x}', \underline{x}'')\\
\mathrm{ACrit}^{\curvearrowleft\!\!\!\curvearrowright}_{R}(\underline{x}, \underline{x}') &\equiv \forall \underline{x}''.\ (\ R(\underline{x}, \underline{x}'') \rightarrow R(\underline{x}', \underline{x}'')\ ) \vee (\ R(\underline{x}, \underline{x}'') \leftarrow R(\underline{x}', \underline{x}'')\ )
\end{aligned}
$$

The equations just captures $\sqsubseteq_{\mathcal{R}}$ and $\equiv_{\mathcal{R}}$ syntactically.

For the lemma we assume that we are in the context of a sequential component, where the state variables $\underline{x}$ are always passed unmodified as additional arguments to (recursive) calls and the formula R is only expressed in the state variables $\underline{x}$. Furthermore, we assume the state variables are never hidden by a **let**-binding in the body of the procedure.

Lemma 5.17 (Atomicity Criteria & (Recursive) Calls)**.** *A (recursive) call* $\mathtt{Proc}(\underline{t}; \underline{x}, \underline{y})$*, where $\underline{x}$ are the state variables of the corresponding component and $\mathit{free}(\underline{t}) \cap \mathit{free}(R) = \emptyset$ and $\mathit{free}(R) \subseteq \underline{x} \cup \underline{x}'$ holds, with declaration*

$$\mathtt{Proc}(\underline{u}; \underline{x}, \underline{z})\ \{\, p \,\}$$

is R-atomic, R-neutral, R-introducible or R-retractable, if the program

$$p' \equiv \textbf{\textit{let}}\ \underline{u} = \underline{t}\ \textbf{\textit{in}}\ \{\, \hat{p}\{\underline{z} \mapsto \underline{y}\} \,\}$$

is, where $\hat{p}$ is derived from p by replacing all recursive calls $\mathtt{Proc}(\underline{t}_0; \underline{x}, \underline{y}_0)$*, with*

$$\textbf{\textit{choose*}}\ \underline{x}_1, \underline{y}_1\ \textbf{\textit{with}}\ \mathit{ACrit}^{\dagger}_{R}(\underline{x}, \underline{x}_1)\ \textbf{\textit{in}}\ \underline{y}_0 \ :=\ \underline{y}_1, \underline{x} \ :=\ \underline{x}_1$$

for the corresponding $\dagger \in \{\curvearrowright\!\!\!\curvearrowleft, \curvearrowleft, \curvearrowright, \equiv\}$. The recursive calls must also satisfy the restriction $\mathit{free}(\underline{t}_0) \cap \mathit{free}(R) = \emptyset$. □

The additional condition $\mathit{free}(\underline{t}) \cap \mathit{free}(R) = \emptyset$ in Lem. 5.17 for the initial call and $\mathit{free}(\underline{t}_0) \cap \mathit{free}(R) = \emptyset$ for all recursive calls essentially states that input and output parameters are always in main memory for the rule to apply.

Note that Lem. 5.17 is a rather coarse abstraction for recursive calls, for example variables in main memory are assigned arbitrarily by it, but it is possible to strengthen the proof obligation with assertions directly after the recursive call and establish them during the invariant proofs of Sec. 4.6 if this should be necessary.

A very strong criterion for atomic blocks is given by Lem. 5.18.

Lemma 5.18 (Atomicity Criteria & Atomic Blocks)**.** *If p is R-atomic, R-neutral, R-retractable or R-introducible, so is* ***atomic*** $\varphi\ \{\, p \,\}$ *for any φ.*

Proof. Note that any execution where φ does not hold initially is a non-termination one in the sequential setting. Every finite execution of **atomic** $\{\, p \,\}$ is either an execution of p or has the form (s_0, s_n) and there is an execution $I = (s_0, \ldots, s_n)$ of p. In the first case the proposition immediately follows. In case of R-neutrality and R-retractability $s_0 \equiv_{\mathcal{R}} s_n$ and $s_n \sqsubseteq_{\mathcal{R}} s_0$ immediately hold, respectively. For R-introducibilty there is an execution $I' = (s_0, \ldots, s_m)$ with $s_n \sqsubseteq_{\mathcal{R}} s_m$. Then (s_0, s_m) is also an execution of **atomic** $\{\, p \,\}$. □

$$\frac{\begin{array}{l}\varphi(\underline{X}), \Box\, \mathit{inv}(\underline{X}), \underline{x} = \underline{X} \\ \vdash \left[\underline{X}'' = \underline{X}',\ \mathit{true},\ \left(\mathrm{ACrit}^{\curvearrowright}_{R}(\underline{x}, \underline{X}) \vee \langle p\{\underline{X} \mapsto \underline{x}\}\rangle\ (\mathrm{ACrit}^{\curvearrowright}_{R}(\underline{X}, \underline{x}))\right),\ p\right]\ \mathit{true}\end{array}}{\varphi, \mathit{inv} \vdash p : \curvearrowright\!\!\!\!\curvearrowright_{R}}\ R\text{-Atomicity}$$

$$\frac{\varphi(\underline{X}), \Box\, \mathit{inv}(\underline{X}) \vdash \left[\underline{X}'' = \underline{X}',\ \mathrm{ACrit}^{\equiv}_{R}(\underline{X}, \underline{X}'),\ \mathit{true},\ p\right]\ \mathit{true}}{\varphi, \mathit{inv} \vdash p : \equiv_{R}}\ R\text{-Neutrality}$$

$$\frac{\varphi(\underline{X}), \Box\, \mathit{inv}(\underline{X}), \underline{x} = \underline{X} \vdash \left[\underline{X}'' = \underline{X}',\ \mathit{true},\ \mathrm{ACrit}^{\curvearrowright}_{R}(\underline{x}, \underline{X}),\ p\right]\ \mathit{true}}{\varphi, \mathit{inv} \vdash p : \curvearrowleft_{R}}\ R\text{-Retractability}$$

$$\frac{\begin{array}{l}\varphi(\underline{X}), \Box\, \mathit{inv}(\underline{X}), \underline{x} = \underline{X} \\ \vdash \left[\underline{X}'' = \underline{X}',\ \mathit{true},\ \langle p\{\underline{X} \mapsto \underline{x}\}\rangle\ (\mathrm{ACrit}^{\curvearrowright}_{R}(\underline{X}, \underline{x}))\,,\ p\right]\ \mathit{true}\end{array}}{\varphi, \mathit{inv} \vdash p : \curvearrowright_{R}}\ R\text{-Introducibility}$$

Figure 5.6: Rely/Guarantee Rules of the Calculus for R-Atomicity, R-Neutrality, R-Retractability & R-Introducibility

Lem. 5.19 proves that assertions may just be assumed for a proof of atomicity.

Lemma 5.19 (Atomicity Criteria & Assertions)**.** *If p is R-atomic, R-neutral, R-retractable or R-introducible in a context where φ holds, so is $\{\, \textbf{\textit{assert}}\ \psi; p \,\}$ in any context.*

Proof. Any termination execution $I = (s_0, \ldots, s_n)$ of $\{\, \textbf{assert}\ \psi; p \,\}$ satisfies $s_0 \models \varphi$. □

Additionally, one can make any R-atomic single-step program q R-introducible by combining it non-deterministically with a R-introducible program p to $\{\, p \vee q \,\}$.

Lemma 5.20 (R-Introducible & Nondeterminism)**.** *If p is R-introducible and has a finite execution starting from any state and q is R-atomic, then $\{\, p \vee q \,\}$ is R-introducible.*

Proof. Any finite execution $I = (s_0, \ldots, s_n)$ of $\{\, p \vee q \,\}$ is either one of p or q. In the first case, there is an R-subsuming execution immediately by R-introducibility of p. Otherwise, either $s_n \sqsubseteq_{\mathcal{R}} s_0$ holds or there already is an R-subsuming execution by R-atomicity of q. In the former case R-introducibility of p yields a R-subsuming execution $I' = (s_0, \ldots, s_m)$ with $s_0 \sqsubseteq_{\mathcal{R}} s_m$. This step assumes that p actually has any finite executions starting in s_0. Then I' is also an R-subsuming execution for s_n, because $s_n \sqsubseteq_{\mathcal{R}} s_0 \sqsubseteq_{\mathcal{R}} s_m$ holds. □

Lem. 5.21 states that in order to prove crash-recover-atomicity, it is sufficient to prove crash-atomicity, which significantly simplifies the reasoning if the recover procedure is given by a large and complex program.

Lemma 5.21 (Crash-Recover-Atomicity)**.** *If a program p is crash-atomic, then it is also crash-recover-atomic.*

Proof. The assumption yields crash-subsuming states. These states are also crash-recover-subsuming by Lem. 5.3. □

5.2.4 R-Atomicity Calculus

Fig. 5.6 reduces the judgments $\varphi, \mathit{inv} \vdash p : \dagger$ for $\dagger \in \{\curvearrowright\!\!\!\!\curvearrowright, \curvearrowleft, \curvearrowright, \equiv\}$ to purely rely/guarantee reasoning. The criteria only talk about intervals with empty environment, therefore the rely states that environment steps do not modify the flexible variables $\underline{X}$. For all rules except *R-Neutral*, the initial state is stored in the static variable $\underline{x}$. The invariants that needs to be

$$\frac{\varphi, inv \vdash p : \dagger_R}{\varphi, inv \vdash p : \curvearrowright_R} \text{ Reduce } R\text{-Atomic}$$

$$\frac{\varphi, inv \vdash p : \equiv_R}{\varphi, inv \vdash p : \curvearrowleft_R} \text{ Reduce } R\text{-Retractable} \qquad \frac{\varphi, inv \vdash p : \equiv_R}{\varphi, inv \vdash p : \curvearrowright_R} \text{ Reduce } R\text{-Introducible}$$

$$\frac{\varphi \wedge \psi, inv \vdash p : \dagger_R}{\varphi, inv \vdash \textbf{atomic } \psi \ \{ p \} : \dagger_R} \ \dagger \in \{\curvearrowright, \curvearrowleft, \curvearrowright, \equiv\} \text{ Atomic} \qquad \frac{\varphi \wedge \psi, inv(\underline{x}) \vdash \langle\!| p |\!\rangle \ true}{\varphi, inv \vdash \textbf{atomic } \psi \ \{ p \} : \curvearrowright_R} \ R\text{-Atomic}$$

$$\frac{}{\varphi, inv \vdash \underline{x} \ := \ \underline{t} : \curvearrowright_R} \ R\text{-Atomic Assignment} \qquad \frac{}{\varphi, inv \vdash \underline{x} \ := \ \underline{t} : \equiv_R} \ \text{RAM Assignment, } \underline{x} \cap free(R) = \emptyset$$

$$\frac{\varphi, inv \vdash p : \curvearrowright_R \quad \varphi, inv(\underline{x}_0), \underline{x}_0 = \underline{x}, \langle p \rangle \ true \vdash [p] \ (\text{ACrit}^{\curvearrowright}_R(\underline{x}_0, \underline{x}))}{\varphi, inv \vdash p : \curvearrowleft_R} \ R\text{-Atomic Retractable}$$

$$\frac{\varphi, inv \vdash p : \curvearrowright_R \quad \varphi, inv(\underline{x}_0), \underline{x}_0 = \underline{x}, \langle p \rangle \ true \vdash \langle p \rangle \ (\text{ACrit}^{\curvearrowright}_R(\underline{x}_0, \underline{x}))}{\varphi, inv \vdash p : \curvearrowright_R} \ R\text{-Atomic Introducible}$$

$$\frac{\varphi, inv \vdash p : \curvearrowright_R \quad inv \vdash q : \curvearrowright_R}{\varphi, inv \vdash \{p; q\} : \curvearrowright_R} \text{ Seq. Comp. Left} \qquad \frac{\varphi, inv \vdash p : \curvearrowleft_R \quad inv \vdash q : \curvearrowright_R}{\varphi, inv \vdash \{p; q\} : \curvearrowright_R} \text{ Seq. Comp. Right}$$

$$\frac{\varphi, inv \vdash p : \dagger_R \quad inv \vdash q : \dagger_R}{\varphi, inv \vdash \{p; q\} : \dagger_R} \text{ Seq. Comp.} \qquad \frac{\varphi \wedge \psi, inv \vdash p : \dagger_R}{\varphi, inv \vdash \textbf{assert } \psi; p : \dagger_R} \text{ Assertion}$$

$$\frac{\varphi \wedge \psi, inv \vdash p : \dagger_R \quad \varphi \wedge \neg \psi, inv \vdash q : \dagger_R}{\varphi, inv \vdash \textbf{if* } \psi \textbf{ then } p \textbf{ else } q : \dagger_R} \text{ If} \qquad \frac{(\varphi \vee \varphi') \wedge \psi, inv \vdash p : \dagger_R}{\varphi, inv \vdash \textbf{while } \psi \textbf{ do } \{ p; \textbf{assert } \varphi' \} : \dagger_R} \text{ While}$$

$$\frac{\varphi, inv \vdash p : \dagger_R \quad \varphi, inv \vdash q : \dagger_R}{\varphi, inv \vdash p \vee q : \dagger_R} \text{ Or} \qquad \frac{inv \vdash p : \dagger_R}{\varphi, inv \vdash p^* : \dagger_R} \text{ Iteration}$$

$$\frac{\varphi, inv \vdash p : \curvearrowright_R \quad \varphi, inv \vdash q : \curvearrowright_R \quad \varphi, inv \vdash \langle p \rangle \ true}{\varphi, inv \vdash p \vee q : \curvearrowright_R} \ \text{Or Left (Or Right symmetric)}$$

$$\frac{\varphi \wedge \psi, inv \vdash p : \dagger_R \quad \varphi \wedge \forall \underline{x}. \neg \psi, inv \vdash q : \dagger_R}{\varphi, inv \vdash \textbf{choose* } \underline{x} \textbf{ with } \psi \textbf{ in } p \textbf{ ifnone } q : \dagger_R} \ \text{Choose } \underline{x} \cap free(R) = \emptyset$$

$$\frac{\varphi, inv \vdash p' : \dagger_R}{\varphi, inv \vdash \texttt{Proc}(\underline{t}; \underline{x}, \underline{y}) : \dagger_R} \ \text{Call } p' \text{ as in Lem. 5.17 on page 56}$$

Figure 5.7: Calculus for R-Atomicity, R-Neutrality, R-Retractability & R-Introducibility for Sequential Programs, with $\dagger \in \{\curvearrowleft, \curvearrowright, \equiv\}$ except where otherwise noted

established then ensures the desired criterion. Note that infinite traces are not excluded from consideration by the rules, i.e., the rules imply a stronger criterion.

Obviously rely/guarantee reasoning is difficult, specifically the existence proof of an R-subsuming execution of p in every intermediate step is quite laborious. Fig. 5.7 shows an alternative proof method. The rules of the calculus profit from the assertions and invariants established as part of the termination proof of Sec. 4.6.

Most rules are proven correct by the above lemmas, except for *RAM Assignment*, R-

Atomic Retractable and *R-Atomic Introducible*. The rule *RAM Assignment* holds, because R is arbitrary on all variables outside $\mathit{free}(R)$. If we choose R as the crash, then $\underline{x} \cap \mathit{free}(R)$ corresponds to the intuition that the variables $\underline{x}$ are in main memory and therefore not mentioned in the formula describing the transition of the crash. Then the values of $\underline{x}$ are irrelevant to the behavior modulo R and the assignment is R-neural. The other two rules underlies the insight that if a program is already R-atomic then it is already clear that the "interesting" behavior is at the initial and final states of the interval. The dynamic logic proof obligations then only need to establish that the behavior can be observed solely in the initial and final states for retractability and introducibility, respectively. For both rules at least one terminating execution may be assumed additionally.

The additional condition $\underline{x} \cap \mathit{free}(R) = \emptyset$ for the rule *Choose* just means that renaming is necessary to avoid clashes and is not a restriction per se.

5.3 Data Refinement

The atomicity refinement of the previous step usually yields a specification component C' for the initial component C as discussed for the example of the Flashix file system in Ch. 6. This step obviously maintains the complex data structures and algorithms of the implementation C.

Like in standard data refinement [40], *data refinement* of crash-aware components allows for changes in the representation of data. It is typically proved using forward simulations. New proof obligations for crash-aware components result from the reset transitions $s_n \xrightarrow{\text{Reset}} s_{n+1}$ and internal operations.

The proof obligations for changing data representation are just slightly more complex than standard data refinement.

Theorem 3 (Data Refinement)**.** *A refinement $A \sqsubseteq C$ of two compatible specification components A and C is implied by a forward simulation $\mathit{abs}(\underline{x}^A, \underline{x}^C)$ satisfying the conditions (1.) to (5.) for all $i \in \mathfrak{I}, k \in \mathfrak{K}$, inputs $\underline{in}$ and outputs $\underline{out}_0$ and $\underline{out}_1$.*

1. $\langle\!|\, \mathtt{init}^C(;\underline{x}^C) \,|\!\rangle \; \big(\; \langle \mathtt{init}^A(;\underline{x}^A)\rangle \; \mathit{abs}(\underline{x}^A, \underline{x}^C) \;\big)$ (initialization)
2. $\mathit{pre}_i^A(\underline{in}, \underline{x}^A) \wedge \mathit{abs}(\underline{x}^A, \underline{x}^C)$ (correctness)
   $\rightarrow \langle\!|\, \mathtt{Op}_i^C(\underline{in}; \underline{x}^C, \underline{out}_0) \,|\!\rangle$
   $\Big(\; \langle \mathtt{Op}_i^A(\underline{in}; \underline{x}^A, \underline{out}_1)\rangle \; \big(\mathit{abs}(\underline{x}^A, \underline{x}^C) \wedge \underline{out}_0 = \underline{out}_1\big) \;\Big)$
3. $\mathit{guard}_k^C(\underline{x}^C) \wedge \mathit{abs}(\underline{x}^A, \underline{x}^C)$ (internal)
   $\rightarrow \langle\!|\, \mathtt{IOp}_i^C(;\underline{x}^C) \,|\!\rangle \; \Big(\langle \mathtt{IOp}_i^A(;\underline{x}^A)\rangle \; \mathit{abs}(\underline{x}^A, \underline{x}^C)\Big)$
4. $\mathit{sync}^A(\underline{x}^A) \wedge \mathit{abs}(\underline{x}^A, \underline{x}^C) \rightarrow \mathit{sync}^C(\underline{x}^C)$ (synchronization)
5. $\mathit{abs}(\underline{x}_0^A, \underline{x}_0^C) \wedge \mathit{crash}^C(\underline{x}_0^C, \underline{x}_1^C)$ (crash)
   $\rightarrow \langle\!|\, \mathtt{recover}^C(;\underline{x}_1^C) \,|\!\rangle$
   $\Big(\; \exists \underline{x}_1^A.\; \mathit{crash}^A(\underline{x}_0^A, \underline{x}_1^A) \wedge \langle \mathtt{recover}^A(;\underline{x}_1^A)\rangle \; \mathit{abs}(\underline{x}_1^A, \underline{x}_1^C) \;\Big)$

Note that for interface operations the precondition of the abstract component A is used. In contrast for internal operations the guard of the concrete component C is assumed.

The synchronization condition states that fewer states of component A are synchronized and therefore all retractions over n operation of C are also allowed for A. The crash condition abstracts the remaining effect of a power cut after a recovery of both components.

Proof of Thm. 3. The proof composes commuting diagrams as usual, starting with two related initial states given by (1.). For transitions of interface and internal operations, proof obligation (2.) and (3.) gives the relevant commuting diagram, respectively. A retraction of

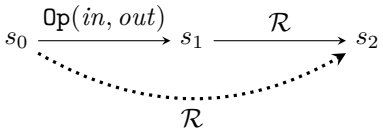

Figure 5.8: R-Retractable Transition

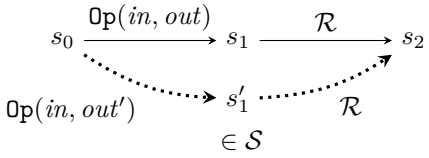

Figure 5.9: S-R-Completable Transition

component C is mapped to a retraction over the same number of operations in A. Condition (4.) ensures that each unsynchronized state retracted by C can be retracted by A as well. The re-execution of the reset transition commutes due to (2.) and (3.). Condition (5.) commutes the rest of the reset transition. □

Note that the invariants of Sec. 4.6 and the corresponding theorems about the operations can be used in the proofs of data refinement.

5.4 Crash Refinement

The third kind of refinement, termed *crash refinement*, replaces the specification of the reset transition of a single specification component, i.e., crash refinement assumes that the data structures and operations are the same on both levels. The basic idea is to move parts of a power cut from a crash transition to the retraction transition by looking at the history fragment $s_n \xrightarrow{\mathtt{Op}_i(in,out)} s_{n+1} \xrightarrow{\text{Reset}} s_{n+2}$ of the component before a reset transition and construct a different explanation of how the component ended up in s_{n+2}. This construction yields an alternative intermediate state s'_{n+1} from a set $\mathcal{S} \subseteq dom(\mathcal{R})$. The relation $\mathcal{R}$ and the formula R denote the crash and subsequent recovery as in Eqn. ($\mathcal{R}$) and Eqn. (R-is-*crash*-`recover`) on page 50. This allows us to simplify the crash and recovery transition to the relation $\mathcal{S} \triangleleft R$, where $\mathcal{S} \triangleleft \mathcal{R}$ denotes the domain restriction of relation $\mathcal{R}$ to the set $\mathcal{S}$. We assume the set of states $\mathcal{S}$ is given by a formula S.

Definition 5.22 (R-Retractable Transition)**.** A transition $s_0 \xrightarrow{\mathtt{Op}(in,out)} s_1$ of an operation `Op` of a specification component is *R-retractable*, if and only if every state s_2 with $s_1 \xrightarrow{\mathcal{R}} s_2$, also satisfies $s_0 \xrightarrow{\mathcal{R}} s_2$.

If a transition is R-retractable, it did not have any immediate permanent effect and we can ignore that it ever took place directly before a crash happened. Figure 5.8 depicts this alternative execution in bold. This does not mean that the execution will never have a permanent effect. Any of the subsequent operations may very well persist the data of previous operations.

Note that Def. 5.22 defined R-retractable for *transitions*, not programs as is done in Sec. 5.2. A program is R-retractable if all state transitions of a finite execution are R-retractable.

Definition 5.23 (S-R-Completable Transition)**.** A transition $s_0 \xrightarrow{\mathtt{Op}(in,out)} s_1$ of an operation `Op` of a specification component is *S-R-completable*, if and only if for every state s_2 with $s_1 \xrightarrow{\mathcal{R}} s_2$ there is an execution $s_0 \xrightarrow{\mathtt{Op}(in,out')} s'_1$ with $s'_1 \in \mathcal{S}$ and $s'_1 \xrightarrow{\mathcal{R}} s_2$ for some output out'.

If a transition is S-R-completable it is possible to construct an alternative partial execution that ended in an $\mathcal{S}$-state without any difference after a crash. Figure 5.9 depicts this alternative execution in bold.

Def. 5.24 lifts Def. 5.22 and Def. 5.23 to the level of one operation.

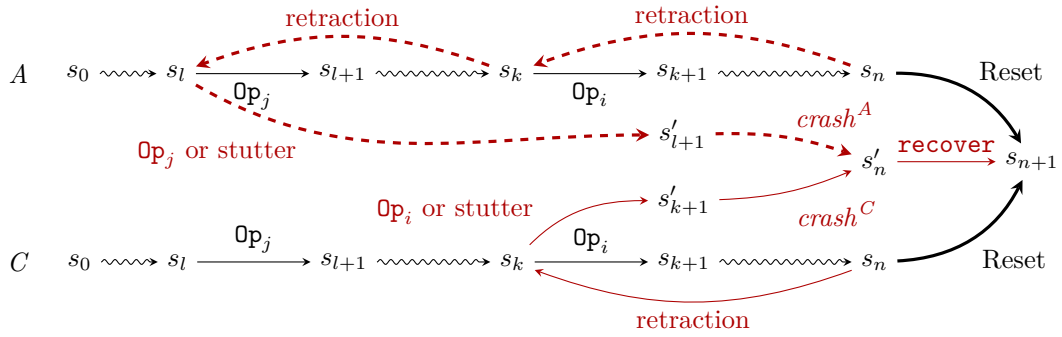

Figure 5.10: From the State-based to the Operations-based Reset Specification

Definition 5.24 (S-R-Completable Operation)**.** An operation $\mathtt{Op}$ of a specification component is *S-R-completable*, if and only if every transition of $\mathtt{Op}$ is either R-retractable or S-R-completable.

The following theorem can be used to abstract a state-based crash specification as part of C to an operations-based crash specification by A.

Theorem 4 (Crash Refinement)**.** *Given two specification components*

$$A = \left(\underline{x}, \mathtt{init}, \{\mathtt{Op}_i\}_{i\in\mathfrak{I}}, \{\mathtt{IOp}_k\}_{k\in\mathfrak{K}}, sync^A, crash^A, \mathtt{recover}, \emptyset\right)$$
$$C = \left(\underline{x}, \mathtt{init}, \{\mathtt{Op}_i\}_{i\in\mathfrak{I}}, \{\mathtt{IOp}_k\}_{k\in\mathfrak{K}}, sync^C, crash^C, \mathtt{recover}, \emptyset\right)$$

that only differ in their crash and synchronized predicates. Then $A \sqsubseteq C$ is implied by the proof obligations (1.) to (3.) for all interface and internal operations $\mathtt{Op} \equiv \mathtt{Op}^C \equiv \mathtt{Op}^A$.

1. $\left(\exists \underline{x}_2.\ crash^A(\underline{x}_0, \underline{x}_2)\right) \wedge crash^C(\underline{x}_0, \underline{x}_1) \rightarrow crash^A(\underline{x}_0, \underline{x}_1)$
2. $sync^A(\underline{x}) \rightarrow sync^C(\underline{x})$
3. $\langle \mathtt{Op}^C(in; \underline{x}_0, out)\rangle\ crash^C(\underline{x}_0, \underline{x}_1)$
   $\rightarrow crash^C(\underline{x}_0, \underline{x}_1) \vee \left(\ \langle \mathtt{Op}^A(in; \underline{x}_0, out')\rangle\ crash^A(\underline{x}_0, \underline{x}_1)\ \right)$

The first proof obligation of Thm. 4 asserts that if the state $\underline{x}_0$ is in the domain of the crash predicate of A, then it corresponds to the crash predicate of C. The second proof obligation ensures that the component A permits further retractions than C. The third condition establishes that all operations are S-R-completable for

$$S \equiv \exists \underline{x}'.\ crash^A(\underline{x}, \underline{x}') \qquad \text{and} \qquad R \equiv crash^C(\underline{x}, \underline{x}'),$$

i.e. S and R corresponds to the domain of the crash predicate of A and the crash predicate of C, respectively.

We usually apply Thm. 4 with crash predicates that satisfy the (stronger) condition that the domain of the crash predicate of A is a proper subset of the domain of the crash predicate of C. Then the theorem strengthens the crash transition, i.e., a crash can happen in fewer states of component A than of component C and is therefore simpler to express. This is compensated by further retractions on the history of A in comparison to those of C. These retractions are then easily propagated upwards over abstractions with data refinement (Thm. 3 on page 59).

Proof of Thm. 4. We choose the run of C as the run of A and focus on the reset transition. Figure 5.10 depicts the situation before the power cut (omitting input and output labels for brevity), starting in a state s_0 where both C and A are synchronized. Such a state exists

because at least in the initial state and after every power cut *both* components are in a synchronized state. The four parts of the reset transition of C are depicted in the figure starting in state s_n and ending in s_{n+1}. First, C retracts all operations and reaches state s_k, then some operation is re-executed and reaches state s'_{k+1} (or the transition stutters). Finally, the crash and recovery are applied and yield the final state s_{n+1}.

We construct a matching reset transition of A, depicted by **arrows - - ➝** in the figure. All operations that C retracts are also retracted by A (*retraction* from s_n to s_k). However, the retraction transition might be further still (*retraction* from s_k to s_l). The idea is to determine a state s'_{l+1} of component A with the properties shown in the figure: there is an additional second retraction to s_l, a re-execution that yields s'_{l+1} and the (restricted) crash $crash^A$ is possible in state s'_{l+1}.

The construction considers the *alternative run*

$$s_0 \rightsquigarrow s_k \xrightarrow{\texttt{Op}_i \text{ or stutter}} s'_{k+1} \xrightarrow{crash^C} s'_n \xrightarrow{\texttt{recover}} s_{n+1}$$

of C that does not retract or re-execute any operations. This run is implied by Remark 4.21 on page 37. The existence of A's retraction and re-execution is proven by induction over k.

If the sequence is empty ($k = 0$ and the transition stutters), then there are only the transitions

$$s_0 \xrightarrow{crash^C} s'_n \xrightarrow{\texttt{recover}} s_{n+1}$$

starting from the synchronized state s_0. By proof obligation (1.),

$$s_0 \xrightarrow{crash^A} s'_n \xrightarrow{\texttt{recover}} s_{n+1}$$

is also a run of A: the additional retraction and re-execution transitions stutter. Note that the crash predicate is applicable in this state, because crashes are always possible in synchronized states according to the definition of weak admissibility of components (Def. 4.3 on page 28).

Otherwise, $\texttt{Op}_i$ is S-R-completable and therefore the transition $s_k \xrightarrow{\texttt{Op}_i} s'_{k+1}$ is either:

- R-Retractable and therefore

  $$s_0 \rightsquigarrow s_k \xrightarrow{crash^C} s'_n \xrightarrow{\texttt{recover}} s_{n+1}$$

  is also a valid run. The induction hypothesis gives a matching run of A and a history jump over m operations for this sequence. The retraction for the original sequence then is over $m + 1$ operations and we take the re-execution from the induction hypothesis which ends in the state s_{n+1}.

- S-R-completable and

  $$s_k \xrightarrow{\texttt{Op}_i} s''_{k+1} \xrightarrow{crash^C} s'_n \xrightarrow{\texttt{recover}} s_{n+1}$$

  holds for some state s''_{k+1} in the domain of the crash predicate $crash^A$. We choose $s'_{l+1} = s''_{k+1}$ and by proof obligation (1.),

  $$s_0 \rightsquigarrow s_k \xrightarrow{\texttt{Op}_i} s'_{l+1} \xrightarrow{crash^A} s'_n \xrightarrow{\texttt{recover}} s_{n+1}$$

  is the alternative run of A with s'_{l+1} in the domain of $crash^A$ and the retraction stutters. □

5.5 Related Work

Refinement and abstraction of atomicity is quite common in the context of the verification of concurrent systems as discussed in Ch. 13 further. Lipon [87] observed that under certain conditions a program can be replaced by an atomic section. The calculus of atomic actions due to Elmas et al. [42] is an extension of Lipton's approach for highly concurrent, linearizable programs. The refinement calculus of Back [14] uses the opposite direction. Instead of abstracting a program to an atomic block, one starts out with an atomic program and splits it into smaller actions.

The Crash Hoare logic employed in the FSCQ file system [27, 28, 26] adds a crash condition to standard Hoare triples. The verification considers each individual step of a program and has to prove that the step implies the crash condition. The calculus for atomicity presented in this chapter allows for an incremental increase in atomicity of an entire component. In the context of the Flashix file system the criterion of crash-introduciblity can be proven by a simple pre/post-verification. This simplifies proofs significantly.

Ntzik et al. [101] propose a variant of separation logic [114] that facilitates reasoning about volatile and durable memory and power failures. The judgments $S \vdash \{P_V|P_D\}\ p\ \{Q_V|Q_D\}$ of the language extend separation logic with a fault-condition S. The formula P_V and P_D describes the volatile and durable part of the state before the execution of p, respectively. The fault-condition S describes the states of durable memory after a recovery. This is similar to the crash condition of Chen et al. [27, 28, 26]. The methodology is similar to the state-based approach, since the fault condition S refers to the state and it is the only guarantee that is given in the event of a power failure. In [101] the methodology is applied to a fault-tolerant bank transfer and to a model of a database system with logging.

Marić and Sprenger [90] model power failures explicitly as exceptions with a handler that performs the recovery and increase the atomicity with respect to power failures incrementally, similar to the approach of this chapter. They also observe that a complete run of a program might already capture the full crash behavior of the system. Their approach is similar to the state-based approach for sequential, crash-aware components.

An implicit specification mechanism with synchronized states and retractions is not considered in related work.

Standard data refinement is due to Hoare et al. [64, 60]. It is used in many formalism for incremental refinement, such as Z [133, 40, 41].

6 The Flashix File System and its Components: Write-Back Caching, Commit & Crashes as Cross-Cutting Concerns

Summary. The development of the Flashix file system is conducted with the formalism and methodology introduced in Ch. 4 and 5. This chapter gives an overview over the components of the file system and details how the methodology is applied to the case study. Two cross-cutting concerns are discussed: The non-local effect of write-back caching of user data by a write-buffer and its propagation with the synchronized states of Ch. 4 is explained conceptually, before the models are presented in more detail in Ch. 11 and Ch. 12. The second aspect that permeates several components is the commit of the file system, at which point several internal data structures are persisted. During normal operations updates to these data structures are cached. Finally, the error model and its interaction with power failures is discussed. The error model facilitates contracting individual steps of a components into larger atomic blocks. This guarantees atomicity with respect to power failures and connects the atomicity refinement developed in Ch. 5 with the presentation of the case study in Chapters 7 to 12.

Publications. This Chapter is based on the overview over the Flashix project published in [118]. The application of the theory to the case study is part of [107, 45]. The models are updated to reflect the current state of the project.

Contents

6.1 Flashix: A Hierarchy of Components

The Flashix file system is composed of a total of 20 components. Fig. 6.1 depicts the components and their relationship. Eleven of the components are specification components. The remaining 9 components are implementations and used for code generation. The implementation components are depicted in gray in the figure. The POSIX specification is realized and implemented in incremental and modular steps, i.e., each of the implementation components deals with a specific concern and delegates conceptually separate issues to a subcomponent. The verification only has to consider one implementation component and an abstract specification of the subcomponent(s) at a time. Correctness of substitution of refined

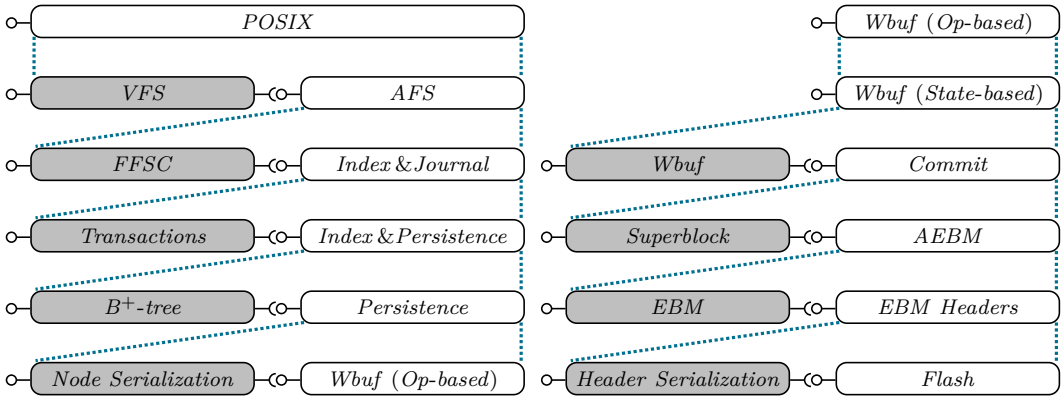

Figure 6.1: Flashix: A Hierarchy of Components

components then ensures that the system composed of only the implementation components is functionally correct, too. The final implementation's observable behavior therefore adheres to the top-level POSIX specification.

In the following the system boundaries and the responsibilities and main data structures of the implementation components are discussed briefly in order to obtain an overview over the interactions between the components. The subsequent chapters 7 to 12 then present the components formally with invariants and abstraction relations and sketch important steps in the refinement proofs. The formal models and mechanized proofs conducted with the interactive verified KIV are available online.[1]

This thesis contributes the following models from Fig. 6.1 and their correctness proofs: The implementation components *Transactions*, *Node Serialization*, *Wbuf*, *Superblock*, *EBM* and *Header Serialization*, and the specification components *Index & Persistence*, *Persistence*, *Wbuf (Op-based)*, *Wbuf (State-based)*, *Commit*, *AEBM*, *EBM Headers* and *Flash*. The specification component *Index & Journal* is joint work with Gidon Ernst [43].

The system boundaries of the Flashix file system are the POSIX specification located at the top-left and the model of flash hardware located at the bottom-right of the figure.

POSIX The top-level component *POSIX* specifies the interface, functional correctness and crash-safety for the entire Flashix file system, in accordance with the POSIX Standard [3]. The component abstractly captures a file system as an algebraic tree that represents the directory structure. The nodes of the tree correspond to files and directories and the edges assign a name to the target node. Structural operations of POSIX modify the directory hierarchy only. The file contents are kept in a separate data structure from the tree, accessed by read, write and truncation operations. Open file handles store the access mode and the current offset in the file. The component and the guarantees Flashix provides is presented in Ch. 7 in more detail.

Flash The lower system boundary of the Flashix file system is the specification *Flash*, which captures the characteristics and limitations of flash hardware. It is based on the Linux interface MTD (= Memory Technology Devices), implemented by all drivers for raw flash hardware. The specification divides the storage into erase blocks and each block into several pages. Preconditions ensure that only sequential writes within an erase block are permissible and that only full pages may be written. The limitation to sequential writes also prevents overwrites of pages until the entire block is erased. The specification also formalizes and clarifies the assumptions of the file system about hardware errors and the behavior of the

[1] http://isse.de/flashix

flash device during power failures. Ch. 8 presents the full model and details and discusses the assumptions underlying the model and therefore Flashix.

The remainder of this section covers the implementation components, in the order from top to bottom of the figure.

Virtual File System Switch (= VFS) The component *VFS* is the first step in the implementation of the POSIX specification and takes on the responsibility of implementing all file system independent operations, such as tree traversal, access checks and the management of open file handles.

For a given path to a file or directory the VFS performs a traversal of the directory tree in order to find the specific file system objects that are accessed. There are three types of file system objects: *Inodes* capture the meta data of files and directories, such as permissions, sizes and number of hard links. *Directory entries* or *dentries* correspond to the edges in the directory tree. A *page* is a segment of the file's contents. The segments are usually of 4KiB size and do not necessarily correspond to the size of a page on the flash hardware. The tree traversal takes permissions checking into account, i.e., in order to access a file or directory certain permissions for all directories on the path are required and need to be checked.

The specification component *AFS* (= Abstract File System) provides access to individual file system objects, i.e., it is possible to retrieve the meta data to files and directories, query whether a directory entry of a given directory exists and which inode it refers to, and read and write individual pages of the file. During path lookup several queries of dentries and inodes are performed, afterwards a call to a single operation modifies the accessed file system objects atomically.

Details about the component *VFS* and *AFS* and the refinement proof $POSIX \sqsubseteq VFS$ are discussed in Ernst [43, Ch. 8] and [48, 46].

Flash File System Core (= FFSC) The component *AFS* is implemented by the Flash File System Core, which introduces the concepts of a log-structured file system. These concepts are necessary in order to deal with the characteristics of flash hardware efficiently. Updates to the file system objects are written out-of-place, and an index is kept in order to locate their most recent version. The file system objects are encapsulated into nodes and several nodes are grouped into an atomic transaction.

Robustness in the event of a power failure is ensured by keeping a log of the changes to the file system since the last commit. During a commit the log is emptied and the index is written to flash. A replay of the log, starting from the flash index, is guaranteed to reconstruct the most recent version of the RAM index in the event of a power failure.

The component *FFSC* introduces the concepts of log-structures file systems on a very abstract level, e.g., the view on the storage device is basically unstructured and maps addresses to algebraic data structures, and delegates their realization to the *Transactions* and B^+*-tree* components. Ch. 11 provides more detail on each of these components.

Transactions The component *Transactions* guarantees atomicity of transactions of several nodes and introduces a block-structured view of the flash device. Out-of-place updates lead to an accumulation of obsolete versions of file system objects in erase blocks. The *Transactions* component locates blocks with few remaining live objects, moves them to a new location and deallocates the erase block. This frees up memory. The component is tightly integrated with the index, in order to check whether a file system object is still in use.

B$^+$-Tree The component *B$^+$-tree* implements the index of the *FFSC* component with a wandering, lazily loaded B$^+$-Tree. The nodes of the tree are stored on flash during the commit. In order to improve performance and decrease memory use not the entire index is written, only the modified parts of the tree "wander" to new locations on the flash device, in order to avoid overwrites and instead perform updates out-of-place. After a reboot or recovery from a power failure, the index is not loaded immediately. Only once a node is accessed during the traversal of the tree, it is loaded from flash. In order to free up space, blocks with obsolete nodes of the B$^+$-Tree also need to be garbage collected.

Node Serialization The transactional journal as well as the B$^+$-Tree persist nodes on the flash device. The component *Node Serialization* is responsible for tracking allocations of erase blocks in the *LEB Property Array*. The data structure also accounts for the number of bytes still referenced by live nodes, i.e., nodes that contain the current version of a file system object or store a node of the B$^+$-Tree. The component serializes both node types to bytes and writes them to flash with the help of a write-buffer. There are two key properties that have to be established by the byte representation of nodes. First, partially written nodes, which might occur due to hardware failures or power loss, must be detected and taken care of in order to ensure that nodes appear to be written atomically to the *Transactions* and *B$^+$-tree* components. Second, if synchronization is requested by the user, the component must be able to write a padding node that fills the remaining space until the write-buffer is properly flushed.

Write-Buffer The component *Wbuf* is the reason for the user-visible crash behavior of the file system. The write-buffer caches the writes to a single erase block. In order to cope with the first limitation of flash hardware, namely the necessity to write sequentially, it provides an operation for appending data to the erase block. The appended data is retained in main memory until a page-aligned write is possible. Only then the pages are written. The write-buffer therefore alleviates two of the characteristics of flash hardware, namely the limitation to sequential and to page-aligned writes.

Such a cache is order-preserving, in the sense that the orders in which data enters and leaves the cache are the same. Therefore, the modeling methodology of synchronized states of Ch. 4 is applicable. The cache stores data of the user, i.e., the directory structure and file contents. Thus, its effect on a power failure is visible across the entire component hierarchy. The theory of Ch. 4 facilitates the propagation of the effect of the cache upwards the entire refinement hierarchy. Sec. 6.2 discusses this issue in more detail.

The write-buffer is the component where crash refinement of Ch. 5 is used to introduce the operations-based view into the component hierarchy.

Superblock The component *Superblock* reserves part of the flash device for several of the file system's internal data structures. The most important one is the superblock, which is used to guarantee the atomicity of file system commits. Furthermore, it stores the LEB Property Array, a pointer to the root node of the index and several others. Each of these data structures is only written during a commit and therefore we refer to them as the *commit data structures*. Persisting every modification to them would incur a huge performance cost by causing a lot of additional I/O operations, and is completely impractical for flash hardware where only out-of-place updates are possible.

A commit also has the advantage that a *consistent* state of these data structures is stored, which facilitates recovery after a reboot or a power failure. It is critical for the consistency of the file system that the commit operation is therefore performed atomically.

To this end the superblock stores the address of the currently valid version of the commit data structures. The commit data structures are written to a secondary location during a commit and only if successful, the superblock is atomically exchanged with the addresses to the new versions.

The components *Node Serialization*, *Wbuf* and *Superblock* are discussed in more detail in Ch. 12. The commit and recovery after a power failure affects multiple components of the hierarchy. Sec. 6.3 gives a brief overview.

Erase Block Manager (= EBM) The erase block manager provides the abstraction of logical erase blocks over the physical erase blocks of the flash hardware. This indirection allows the component to move the contents from one physical block to another, again, transparently to its clients. Data is moved in order to distribute erases of blocks evenly among the physical blocks of the device, thereby increasing its lifetime. In order to pick suitable blocks for wear-leveling several additional internal data structures are kept.

The component also performs a lot of error recovery. If a hardware error, such as an I/O error during reading or writing, is detected the component attempts to move the contents of the respective block to a new location transparently to the user. Errors are not completely masked and may still surface at the level of the client. In these cases, however, the errors can be treated as fatal and a file system should switch to a read-only mode.

The component maintains a partial forward mapping from logical to physical blocks in main memory, and an inverse mapping in each physical block. The forward mapping constitutes a write-back cache, i.e., updates to the inverse mapping are applied only lazily in order to improve performance. After a power failure it is critical that a consistent forward mapping is restored and that no data is lost.

The component *EBM* persists its internal data structures, i.e., mainly the inverse mapping, with the help of the component *Header Serialization*. The component provides an abstract algebraic view over the first few pages of each erase block in order to decouple the serialization of data structures from the algorithmic part of the erase block manager. This facilitates the verification of the erase block manager.

The component *EBM* and *Header Serialization* are discussed in Ch. 10 in more detail. Ch. 14 extends the implementation and verification to a concurrent setting.

6.2 Non-Local Effects of Write-Buffering

This section discusses the effect of the write-buffer component on the entire model hierarchy above this component. The synchronized states of Ch. 4 are employed to uniformly and easily express the effect and propagate it upwards the refinement hierarchy.

As shown by Fig. 6.1 the write-buffer is in the middle of the component hierarchy of the Flashix file system. Flash hardware only supports writing *entire* pages *sequentially* within an erase block. Additionally, overwriting is not easily possible. Therefore it is necessary to cache at least the contents of one flash page in main memory and only perform an I/O operation once a page boundary is crossed.

Fig. 6.2 depicts the same situation at different levels of abstraction, starting from the flash device at the lowest level and up to the level of POSIX at the top. At each level the name of the corresponding component of Fig. 6.1 is shown. In the following we will see how at each of these levels previously completed operations and their written data elements are affected by a power failure.

The figure visualizes an erase block composed of eight pages at the bottom. The first five pages are already filled with data. The sixth page is not yet written. However, the write-buffer above the sixth page already contains some data—visualized in **blue**—for the page. Once the page-sized cache is filled, the sixth page is written and the buffer is moved further along the erase block. If the erase block itself is completely full, the write-buffer is moved to the next block.

The buffer caches user data, such as the directory structure and the file contents, across several operations. The figure visualizes the five write operations W1-W5 requested by the client component of the write-buffer directly above the erase block. The dots signify the

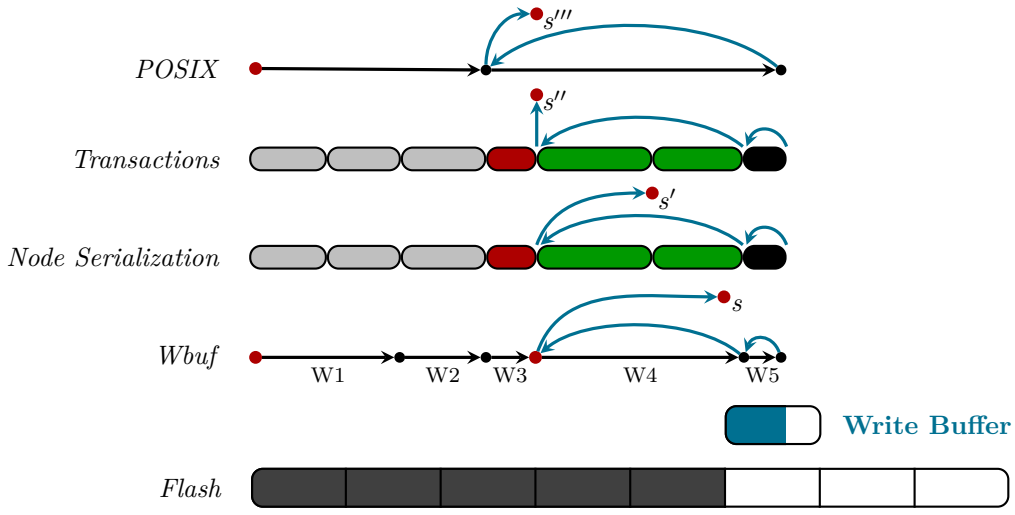

Figure 6.2: The effect of write-back caching of user data is propagated upwards the component hierarchy by synchronized states.

states in between write operations. The starting point of each **black arrow** denotes the offset in the erase block of the write operation. The arrow's length corresponds to the number of bytes written.

In the event of a power failure all data cached in the write-buffer is lost. With the theory discussed in Ch. 4 this is expressible in terms of synchronized states as shown by Equation (wbuf-synchronized). This definition marks all states in between operations where the write-buffer was empty as synchronized.

synchronized states $wbuf.\texttt{bytes} = 0$ (wbuf-synchronized)

Note that Equation (wbuf-synchronized) only illustrates the concepts, Ch. 12 provides the details of the write-buffer component.

In the figure synchronized states are visualized as slightly larger, **red** dots. The synchronized states are exactly above the page boundaries. The effect of a power failure is visualized in the figure by **blue arrows**, i.e., the last operation W5 is retracted and the second to last operation W4 is re-executed. The re-execution yields the synchronized state s at the boundary between the fifth and sixth page of the erase block. Note that the state s reached by a crash can not be explained by the states of the write-buffer component in between operations W1-W5 alone, i.e., re-execution of operations is necessary for an operations-based explanation of the state s.

At the next level of abstraction in Fig. 6.1, the component *Node Serialization* serializes nodes into bytes. A individual node is visualized in the figure as a rounded rectangle. Each of the nodes corresponds to a modification of one file system object, such as directory or file meta data, a directory entry or a segment of a file.

Several nodes are written at once, in a single transaction, in order to minimize the number of I/O operations necessary. The nodes that belong to the same operation of the component *Node Serialization* are visualized in the same color in Fig. 6.2. For example the three **gray** nodes correspond to one operation call to the component *Node Serialization* by its client, and the component issues the two writes W1 and W2 to persist the nodes.

At this level of abstraction a power failure removes the last two nodes, which again corresponds to a retraction of the last, **black** operation and a re-execution of the second to last, **green** operation reaching the state s'. In the state s' only the five nodes are present. The re-execution persists one node and then fails with an error. Note that the second **green**

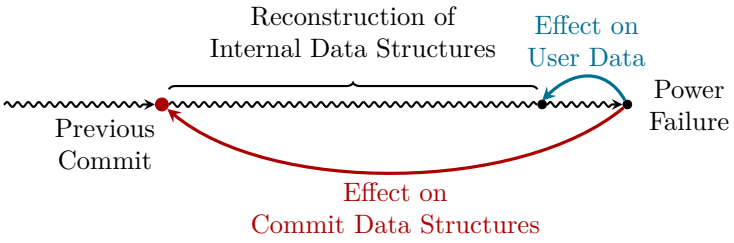

Figure 6.3: Effects of Write-Back Caching of Internal Data Structures
Effects of write-back caching of internal data structures diverges from effects on user data.

node is in reality written partially. Therefore, the component has to detect such partial nodes and must hide them from its clients, i.e., the byte representation of nodes must ensure that partially written nodes are detectable, as discussed in Ch. 12 in more detail. The synchronized states on this level are no longer expressed in terms of the regular state, i.e., in terms of nodes. They are modeled by an explicit state variable *synced*, which is set by a successful call to an explicit synchronization operation and reset by any other operation.

state $synced : \mathbb{B}$ (nodes-synchronized)
synchronized states $synced$

The next level of abstraction, the component *Transactions* groups several nodes into one transaction. The **gray nodes**, the single **red node**, the two **green nodes** and the single **black node** each correspond to one transaction. The transaction is written by one call to the component *Node Serialization*. It groups several modifications of different file system objects. Transactions are atomic with respect to power failures. As shown in the figure, the component *Transactions* retracts both the **black** and **green** operations during a crash. The final state is then the synchronized state s''. In order to guarantee atomicity of transactions the component *Transactions* has to detect and discard incomplete transactions as shown in Ch. 11.

At the top-level the first transaction might correspond to a single POSIX operation while the remaining three transactions are also part of a single operation. The power failure corresponds to a retraction and re-execution of the last operation, yielding the synchronized state s'''.

At the level of component *Transactions* and *POSIX* the synchronized states are also expressed by an additional flag, similar to equation (nodes-synchronized). For these components it is no longer easily feasible to express the crash as a predicate in the state before and after the power failure without a lot of additional specification overhead.

This section shows the main levels of data abstraction above the write-buffer component and how these components are affected by the caching mechanism it employs. As can be seen power failures and recovery are cross-cutting concerns, i.e., it is not easily possible to isolate and encapsulate them in some component. Instead most components of the Flashix file system are affected by power failures and play a part in the recovery, because each of them needs to mask part of the effect of a crash in order to guarantee the desired atomicity with respect to power failures. Synchronized states are a simple approach to uniformly capture the effect of *order-preserving caches* on large hierarchy of components. Order-preserving means that the cache performs I/O operations in the order the data arrives in the cache.

6.3 Commit & Recovery as Cross-Cutting Concerns

The effect on user data as described in the previous section is, however, not the entire effect of a power failure for intermediate components. Every file system has internal, persistent data structures such as an index, which maps identifiers of file system objects to the physical

address of the contents of the latest version of the object. Persisting every modification of these data structures would greatly impact performance. Therefore, internal data structures are written to the flash medium only at specific points, called *commit*. On a power failure the version of these commit data structures at the point of the last commit is therefore recovered.

Fig. 6.3 visualizes the effect of the write-buffer on a power failure as the **blue arrow**. The state of the previous commit is synchronized. The effect of the write-buffer reverts a small number of operations, because only one page is cached. The effect on the internal data structures, however, is quite large and depicted by the **red arrow** in the figure. In order to restore a state where the internal data structures match the user data, a reconstruction mechanism replays the modifications performed between the previous commit and the power failure. Note that this reconstruction is not perfect, i.e., some effects are visible locally. The commit data structures are accessed by different components of the Flashix file system, therefore several of the components in the middle of the model hierarchy are affected by these effects as discussed in more detail in Ch. 11 and Ch. 12. This effect is kept invisible to the user.

The effects are modeled explicitly with the crash predicate of Ch. 4.

6.4 Error Model & Atomicity

Ch. 4 defines a calculus that facilitates reasoning about power failures by substituting an atomic block **atomic** $\{\, p \,\}$ for a subprogram p if p is crash-recover-atomic. We show for every component of the hierarchy depicted in Fig. 6.1 on page 66 that every operation is crash-introducible, which is a stronger criterion.

The hierarchy of components follows the pattern M—©—A where M only contains RAM state, i.e., the crash predicate $crash^M$ of the individual component M is just equivalent to *true*. The entire persistent state is captured by the subcomponent A. The calculus in Fig. 5.7 on page 58 can be employed to prove that all operations of the component M are crash-introducible and by the rules of the calculus it is sufficient to prove that all operations of A are crash-introducible as summarized by Thm. 5.

Theorem 5 (RAM Components & Atomicity)**.** *All operations of a RAM component M with subcomponent A are crash-introducible (and thereby crash-atomic) if all operations of A are crash-introducible.*

Proof. The calculus of Fig. 5.7 on page 58 only yields the crash-introducibility of calls to the subcomponent A as the only nontrivial premises. The reason is that the variables accessed by component M, outside of the subcomponent calls to A, are disjoint from those of the combined crash predicate $\widehat{crash}^M$. Therefore, the rule *RAM-Assignment* can be applied to all assignments of the component M outside the calls of A. □

Fig. 6.4 visualizes how Thm. 5 lifts this insight from the component A to its client component. The run of the client component M is depicted as solid arrows at the bottom. It starts in the state $ms_0 \oplus as_0$ and ends in state $ms_n \oplus as_n$. The large dots indicate that these states are not intermediate state, but states in between two operations of M. Component M calls two operations on the component A before reaching the final state of the operation. According to the Def. 5.10 on page 54 we have to show that for each intermediate state $s_i = ms_i \oplus as_i$ of the operation of M there is another final state $s_m = ms_m \oplus as_m$ of the operation, which subsumes the crash behavior of s_i. For the intermediate state s_i directly after the execution of $\mathtt{Op}_i^A$ in the figure, which produces state $ms'_i \oplus as'_i$ after a crash, we try to complete the operation of M starting in the intermediate state s_i with a final state $ms_m \oplus as_m$ that satisfies $as_m \xrightarrow{crash^A} as'_i$. This constructed final state $ms_n \oplus as_n$ is an alternative explanation of the actual state $ms'_i \oplus as'_i$ after the crash. Notice that the state of component M is irrelevant with respect to the crash, since the crash predicate $crash^M$ is equivalent to *true*. The figure shows an example for such a completion. All RAM operations in

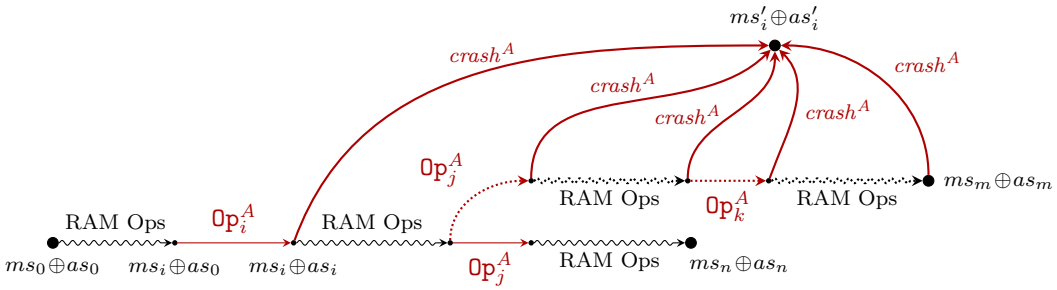

Figure 6.4: Hardware Errors and Crash-Recover-Introducible Completions

between the two subcomponent calls are reused as is. After the crash-introducible execution of $\mathtt{Op}_j^A$ the new run might diverge from its origin. However, it is always maintained that an edge $crash^A$ to the actual state $ms'_i \oplus as'_i$ after the power failure exists. The new run might produce a different call sequence for the subcomponent A, shown by the additional call to $\mathtt{Op}_k^A$ in the figure. This could for example be the effect of a retry mechanism of the component M.

At each level of the hierarchy depicted in Fig. 6.1 on page 66 we then use Thm. 5 in order to conclude that all operations are atomic with respect to power failures. The implementation components depicted in gray in the figure are all RAM components. All specification components are atomic, and only a dynamic logic proof obligation remains to be proved for each of their operations individually, as stated by Thm. 6.

Theorem 6 (Crash-Introducible Specification Components)**.** *An operation* $\mathtt{Op}_i^A$ *of a specification component* A *with state* $\underline{x}$*, precondition* $\mathtt{pre}(\underline{in}, \underline{x})$*, invariant* $\mathtt{inv}(\underline{x})$ *and crash predicate* $crash(\underline{x}, \underline{x}')$ *is crash-introducible if the proof obligation*

$$\begin{aligned} & \mathtt{pre}(\underline{in}, \underline{x}) \wedge \mathtt{inv}(\underline{x}) \wedge crash(\underline{x}, \underline{x}') \\ \rightarrow\ & \langle \mathtt{Op}_i^A(\underline{in}; \underline{x}, \underline{out}) \rangle\, crash(\underline{x}, \underline{x}') \end{aligned}$$

holds for all inputs $\underline{in}$*, outputs* $\underline{out}$ *and states* $\underline{x}$ *and* $\underline{x}'$*.*

Proof. A call $\mathtt{Op}_i^A(\underline{in}; \underline{x}, \underline{out})$ is equivalent to $p \equiv \textbf{atomic}\ \{\, \mathtt{Op}_i^A(\underline{in}; \underline{x}, \underline{out}) \,\}$, because A is a specification component. Application of the rules *R-Atomic Introducible* and *R-Atomic* of the calculus of Fig. 5.7 on page 58 on p yields the above proof obligation as a result, if the formula for R-subsumption is expanded, and additionally the termination condition

$$\mathtt{pre}(\underline{in}, \underline{x}) \wedge \mathtt{inv}(\underline{x}) \rightarrow \langle\!| \mathtt{Op}_i^A(\underline{in}; \underline{x}, \underline{out}) |\!\rangle\, true$$

which is already proven for the component A in its invariant proof of Sec. 4.6. □

The underlying reason why all components are crash-introducible lies in the error model. Every operation of the component *Flash* basically either performs the operation successfully or fails with an error code and without changing the state. Program (error model) shows this pattern.

$$\{ \cdots; err \ :=\ \mathtt{ESUCCESS} \} \vee \{ err \ :=\ \mathtt{EIO} \} \qquad \text{(error model)}$$

All operations following this pattern are immediately crash-introducible by choosing the program on the right-hand side of the nondeterministic choice, see also Lem. 5.20 on page 57. Since error code *err* is in volatile state and therefore arbitrarily modified by a crash, the flash operation has a run, which is completely reverted by a power failure.

Note that Program (error model) does not mean that at each intermediate level of abstraction all operations may fail in this way. At intermediate level components usually

also expose operations that are purely performed in main memory in the implementation and therefore do not return an error code at all. In these circumstances however, the crash predicate reverts the changes performed by the operation and the proof obligation of Thm. 6 is not trivial and reasoning about the specification of the power failure is necessary. An example is the `aebm_unmap` procedure of the component *AEBM*, discussed in Ch. 10 in more detail.

7 The POSIX Specification: Functional Correctness & Crash-Safety of File Systems

Summary. This chapter uses the framework of crash-aware components of Ch. 4 to specify the behavior of a file system. As a basis the informal POSIX specification is used and formalized as a component with precise and succinct guarantees in the event of power cuts or other fatal crashes. The model of POSIX also gives the strongest possible guarantee for other errors during the normal operations of the file system, i.e., an error leaves user-observable state unmodified. The Flashix file system is a refinement of this specification, implementing the behavior demanded by POSIX on top of flash memory, which is known for its error-prone nature.

Publications. The basis of the POSIX specification described in this chapter is published in [49]. The publications [47] and [107] extend the component *POSIX* to cover crash-safety.

Contents

Note that this chapter is based on previous work by Ernst [43, Ch. 7]. The contribution of this thesis is the generalization of the crash behavior with synchronized states. The material is presented to clearly state the correctness criterion achieved by the Flashix file system with respect to errors and power failures, both of which are concerns that pervade the entire file system, and to introduce several concepts of file systems.

7.1 The Directory Tree, File Store and File Handles

The POSIX standard [3] provides an API for C programs that does not only cover file system operations but also e.g. process and thread management and interprocess communication. It is not only supported on Unix-like operating systems, such as Linux, but also on Windows via the *Microsoft POSIX subsystem*.

In POSIX the file system essentially consists of a tree as shown in Fig. 7.1. The nodes of the tree represent files and directories. The edges connect a directory to one of its entries. The name of the directory entry is given by the label of the edge. The root of the tree represents the root

directory "/". The labeling of the path from the root to a file or directory node, separated by "/", is the (or more precisely one) absolute path of this file or directory in the file system.

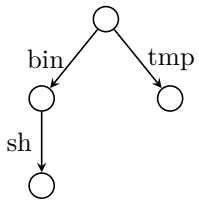

Figure 7.1: The File System Tree

One distinguishing feature of POSIX are *hard links* to files. A hard link allows two paths to refer to the same file, i.e., its meta data such as the size and permissions and contents, by different names. Incremental backups for example copy the changed files only and store a hard link to the previous backup for files that were not altered. This saves a lot of space while still maintaining a complete directory tree of the files that need a backup. From a modeling perspective taking hard links into account requires one indirection between the node in the directory tree that represents a file and the meta data and contents of the file. This indirection are *file identifiers* and are visualized in Fig. 7.2. The two **red arrows** denote two hard links to the same file from different paths in the tree.

This indirection is also necessary to express *orphaned files*. These files are still opened, but have already been deleted from the directory tree, i.e., the file is no longer reachable by any path from the root directory. Reading and writing of the file is, however, still possible through its *file handle*. The **blue arrow** in the figure visualizes an orphaned file, since there is no corresponding arrow from the directory tree pointing to it. Orphaned files are useful for system-wide updates of packages. A process might still refer to old versions of executable files or might still have configuration files opened while these file are being replaced by a new version. The file system needs to recognize orphaned files and delete them during recovery from a power failure or during a normal reboot. Otherwise, precious capacity is lost by storing inaccessible files.

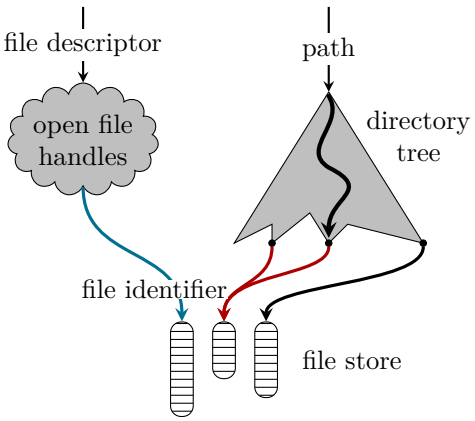

Figure 7.2: Structure of the POSIX File System Interface

With these two features in mind the state of the component *POSIX* is modeled as a tree *root*, a file store *fs* for the file meta data and contents and file handles *ofh* for all opened files.

state $root\colon \texttt{Tree},\quad fs\colon \mathbb{F} \rightarrow\!\!\!\!\!\!\rightarrow \texttt{file-data},\quad ofh\colon \mathbb{D} \rightarrow\!\!\!\!\!\!\rightarrow \texttt{file-handle},\quad sync\colon \mathbb{B}$

The synchronized states of the component *POSIX* are expressed explicitly as an additional state variable *sync*, which is set if the user requests synchronization of the data to the storage device and reset by operations that alter the state but are not persisted immediately and are cached in main memory only.

A directory tree `Tree` is either a file node that refers to a *file identifier* of type $\mathbb{F}$ in the file store *fs* or a directory with meta data and a subtree for each of the directory's entries. The entries are given by their file and directory name only.

data type $\texttt{Tree} = \texttt{filenode}(\texttt{fid}\colon \mathbb{F})$
$\mid \texttt{directorynode}(\texttt{meta}\colon \texttt{metadata}, \texttt{entries}\colon \texttt{String} \rightarrow\!\!\!\!\!\!\rightarrow \texttt{Tree})$

Meta data is an abstract type that only provides three predicates `pr`, `pw` and `px`, which return whether a specific user has read, write and execute permissions to the directory or file, respectively. For example $\texttt{px}(user, meta)$ returns true if and only if the user *user* has execute access to the file or directory with the given meta data *meta*. Execute permissions for a directory are necessary in order to access any child directory or file.

A path into a tree is just a sequence of names of files and/or directories. The path separator "/" is implicitly added in between the path segments of a list.

type alias $\texttt{Path} \equiv \texttt{List}\langle\texttt{String}\rangle$

A path p is considered valid for tree t, written $p \in t$, if it is possible to walk the path p starting from the root of the tree, i.e., if a path in the file system tree exists that has a labeling corresponding to p. If a path p is valid for t, $t[p]$ extracts the subtree of t at path p and $t[p, t']$ replaces the subtree at path p by the tree t'. Removal of the subtree at a given path p is denoted by t -- p.

The file store *fs* maps file identifiers to their meta data and contents. The contents are stored as an array of bytes and the length of the array corresponds to the file's size.

data type $\texttt{file-data} = \texttt{mkfile}(\texttt{meta}\colon \texttt{metadata}, \texttt{content}\colon \texttt{Array}\langle\texttt{Byte}\rangle)$

The open file handles *ofh* map a *file descriptor* of type $\mathbb{D}$, to the file identifier of the opened file, the access mode and the current offset (in bytes) for reading and writing in the file.

data type $\texttt{file-mode} = \texttt{MODE\_R} \mid \texttt{MODE\_W} \mid \texttt{MODE\_RW}$

data type $\texttt{file-handle} = \texttt{mkfilehandle}(\texttt{fid}\colon \mathbb{F}, \texttt{mode}\colon \texttt{file-mode}, \texttt{pos}\colon \mathbb{N})$

The component *POSIX* has two invariants. The first invariant (root-is-dir) states that the root of the tree is always a directory.

invariant $root.\texttt{dir?}$ (root-is-dir)

The second invariant (fids-cons) ensures that the file identifiers referred to by the directory tree *root* and the open file handles *ofh* are valid and together comprise the domain of the file store *fs*.

invariant $dom(fs) = root.\texttt{fids} \cup ofh.\texttt{fids}$ (fids-cons)

The functions $\texttt{fids}\colon \texttt{Tree} \to \texttt{Set}\langle\mathbb{F}\rangle$ and $\texttt{fids}\colon (\mathbb{D} \nrightarrow \texttt{file-handle}) \to \texttt{Set}\langle\mathbb{F}\rangle$ return the set of file identifiers referred to by the corresponding data structure and are defined by

$$ofh.\texttt{fids} = \{\, ofh[fd].\texttt{fid} \mid fd \in dom(ofh) \,\}$$

and

$$\texttt{filenode}(fid).\texttt{fids} = \{\, fid \,\}$$

$$\texttt{directorynode}(meta, entries).\texttt{fids} = \bigcup_{s \,\in\, dom(entries)} entries[s].\texttt{fids}$$

In the actual POSIX interface file identifiers $\mathbb{F}$ and file descriptors $\mathbb{D}$ are identified with natural numbers, i.e., $\mathbb{F} = \mathbb{D} = \mathbb{N}$ holds. For clarity of the presentation these two concepts are separated into distinct types in the presentation.

7.2 Operations & Error Handling

Initialization creates a root directory for some given meta data with the proper permissions and access times set. The file store does not contain any files and no file handles exist yet. Initialization also has to yield a synchronized state, which is achieved by setting the state variable *sync* to *true*.

initialization

$\texttt{posix\_init}(meta\colon \texttt{metadata})$

$root \coloneqq \texttt{directorynode}(meta, \emptyset), \quad fs \coloneqq \emptyset, \quad ofh \coloneqq \emptyset, \quad sync \coloneqq true$

All operations of the component *POSIX*, except for `posix_read` and `posix_write` have the precondition *true*, since the component has to deal with any bogus (but well-typed) input by returning an error code instead. Error codes are for example returned if the file a user tries to open does not exist or if the user does not have the necessary access permissions. For the procedures `posix_read` and `posix_write` it is assumed that the number of bytes requested does not exceed the length of the buffer passed to the procedure.

POSIX error codes fall into three categories. The error code `ESUCCESS` is returned if the operation was successful, high-level errors are returned if additional prerequisites for a successful execution of the operation, such as adequate permissions to the accessed files and directories, are not satisfied. Low-level errors may be returned by any operation if the hardware or system cannot satisfy the request for some "internal" reason.

data type	`Error` =	`ESUCCESS`	success
		| `EEXISTS` | `EISDIR` | `EACCESS` | ⋯	high-level error
		| `EIO` | `ENOMEM` | `ENOSPC` | ⋯	low-level error

For example `EEXISTS` is a high-level error returned by the file creation operation if the file already exists. Low-level errors include input/output error `EIO`, which is caused by unreadable sectors or blocks, the out-of-space errors `ENOSPC` and `ENOMEM`, which are returned when the medium has no free space to store the data and when there is not enough main memory, respectively. The predicate $\lfloor . \rfloor$ denotes whether an error is a low-level error or not.

The specification of the POSIX operations uses convention (error-handling) for the choice of the returned error code. First, an error code is chosen, dependent upon the input of the operation and the state. If preconditions are not met, then the corresponding error(s) are allowed. Low-level errors are always allowed by the predicate `op-pre`. The nondeterminism in the choice of error codes is necessary, because the POSIX component cannot foresee the exact error that may be returned by a file system or flash hardware. Only in the case of `ESUCCESS` the actual code is executed, otherwise the state is left unchanged.

`posix_op`(*in*; *out*, *err*) (error-handling)
  choose *err′* **with** `op-pre`(*in*, *root*, *fs*, *ofh*, *out*, *err′*) **in**
    err := *err′*;
    if *err* = `ESUCCESS` **then**
      ⋯

Note that leaving the user-visible state unmodified is a very strong guarantee and the Flashix file system provides several mechanisms on different layers to ensure that this property is satisfied. For example, the component *Transactions* in Fig. 6.1 on page 66 provides transactions grouping versions of several file system objects into one atomic entity. The component *Node Serialization* is responsible for ensuring that writing a new version of one file system object itself is seen by its client as one atomic operation. These mechanism are discussed in Ch. 11 and Ch. 12 in more detail.

Fig. 7.3 depicts the interface operations of the component *POSIX* that either change or query the structure of the directory tree or ensure synchronization to the persistent medium. Note that POSIX is a specification component and therefore all operations should be interpreted as being enclosed by an atomic section, although the atomic block is omitted for brevity. Error handling is omitted and follows the convention (error-handling).

Note that all operations that alter the persistent state reset the *sync* flag and only the operation `posix_sync` sets it. The operation ensures that the file contents and meta data and the directory tree is completely written to the storage device.

The operation `posix_mkdir` creates a new, empty directory at the given path, if the path does not yet exist. Deletion of a directory has as a prerequisite that the directory is empty, then it just removes the subtree at the path from the directory tree. A directory listing is requested via `posix_readdir` and returns the set of names of the entries.

interface operations

`posix_mkdir`(*path*, *meta*; *err*)
  root[*path*] := `directorynode`(*meta*, ∅), *sync* := *false*

`posix_rmdir`(*path*; *err*)
  root := *root* -- *path*, *sync* := *false*

`posix_readdir`(*path*; *entries*, *err*)
  entries := *dom*(*root*[*path*].`entries`)

`posix_create`(*path*, *meta*; *err*)
  choose *fid* **with** *fid* ∉ *dom*(*fs*) **in**
    root[*path*] := `filenode`(*fid*), *fs*[*fid*] := `mkfile`(*meta*, []), *sync* := *false*

`posix_link`(*srcpath*, *dstpath*; *err*)
  root[*dstpath*] := `filenode`(*root*[*srcpath*].`fid`), *sync* := *false*

`posix_unlink`(*path*; *err*)
  let *fid* = *root*[*path*].`fid` **in**
    root := *root* -- *fid*, *sync* := *false*;
    if *fid* ∉ *root*.`fids` ∪ *ofh*.`fids` **then**
      fs := *fs* -- *fid*

`posix_rename`(*srcpath*, *dstpath*; *err*)
  let *srctree* = *root*[*srcpath*], *exists* = *dstpath* ∈ *root*, *dsttree* = *root*[*dstpath*] **in**
    root := (*root* -- *srcpath*)[*dstpath*, *srctree*], *sync* := *false*;
    if *exists* ∧ ¬ *dsttree*.`dir`? ∧ *dsttree*.`fid` ∉ *root*.`fids` ∪ *ofh*.`fids` **then**
      fs := *fs* -- *dsttree*.`fid`

`posix_sync`(; *err*)
  sync := *true*

Figure 7.3: Structural Operations of the Component *POSIX* (error handling omitted)

File creation allocates a new file identifier and then adds an empty file to the file store and a new file node pointing to it to the directory tree. The operation `posix_link` creates a hard link of a file by adding a file node at the destination with the file identifier of the source. A file can be removed with the operation `posix_unlink`. If this is the last link of the file from the directory tree or from the open file handles, then the file's contents are removed since they can no longer be accessed by the user.

A file or directory can be moved to a new path by `posix_rename`. Note that the destination may already exist if we are renaming a file. In this case the destination file is removed from the tree. This special case of `posix_rename` facilitates replacing the entire contents of a file A *atomically* by first creating and writing a new file B and then renaming B to A.

The file operation of POSIX are shown in Fig. 7.4. Opening and closing of a file adds and removes the file handle, respectively. If the file becomes an orphan after removal of the file handle, it is also deleted from the file store, analogously to the operation `posix_unlink`. Truncation resizes the file to the desired size. If the new size is larger then all bytes above the previous file size are initialized with 0.

Reading and writing takes place at the current file position stored in the file handle and might read and write a smaller number of bytes $count'$. The value is chosen nondeterministically. Note that if $count'$ is less than the requested number of bytes this still counts as a successful operation. Reading just copies the requested bytes from the file contents in the file store *fs* to the given buffer. Writing has to extend the file with bytes initialized to 0 if the current file position is beyond the file's size. Afterwards, the buffer is copied into the file's contents.

The current file position is set with the procedure `posix_seek`. Its flag has one of the following values: The seek flag determines whether the file position is set relative to the offset 0 (`SEEK_SET`), the current file position (`SEEK_CUR`) or the file size (`SEEK_END`). Reading and

interface operations

`posix_open`(*path*, *mode*; *fd*, *err*)
  choose fid_0 **with** $fid_0 \notin dom(ofh)$ **in**
    $ofh[fid_0] \coloneqq$ `mkfilehandle`$(fid, mode, 0)$, $fid \coloneqq fid_0$

`posix_close`(*fd*; *err*)
  let $fid = ofh[fd]$.`fid` **in**
    $ofh \coloneqq ofh$ -- fid
    if $fid \notin ofh$.`fids` $\cup$ $root$.`fids` **then**
      $fs \coloneqq fs$ -- fid

`posix_truncate`(*path*, *filesize*; *err*)
  let $fid = root[path]$.`fid` **in**
    $fs[fid]$.`content` $\coloneqq$ `resize`$(fs[fid]$.`content`$, filesize)$, $sync \coloneqq false$

`posix_read`(*fd*; *buf*, *count*, *err*)
  pre $count \leq \# buf$
  let $fid = ofh[fd]$.`fid` **in**
    choose $count'$ **with** $count' \leq count \wedge count' + ofh[fd]$.`pos` $\leq \# fs[fid]$.`content` **in**
      $buf \coloneqq$ `copy`$(fs[fid]$.`content`$, ofh[fd]$.`pos`$, buf, 0, count')$, $count \coloneqq count'$,
      $ofh[fd]$.`pos` $\coloneqq ofh[fd]$.`pos` $+ count'$
    ifnone
      $count \coloneqq 0$

`posix_write`(*fd*, *buf*; *count*, *err*)
  pre $count \leq \# buf$
  let $fid = ofh[fd]$.`fid` **in**
    choose $count'$ **with** $count' \leq count$ **in**
      $fs[fid]$.`content` $\coloneqq$ `splice`$(buf, 0, fs[fid]$.`content`$, ofh[fd]$.`pos`$, count')$, $count \coloneqq count'$,
      $ofh[fd]$.`pos` $\coloneqq ofh[fd]$.`pos` $+ count'$, $sync \coloneqq false$

`posix_seek`(*fd*, *flag*; *pos*, *err*)
  if $flag =$ `SEEK_CUR` **then** $pos \coloneqq pos + ofh[fd]$.`pos`;
  if $flag =$ `SEEK_END` **then** $pos \coloneqq pos + \# fs[ofh[fd]$.`fid`$]$.`content`;
  $ofh[fd]$.`pos` $\coloneqq pos$

`posix_readmeta`(*path*; *meta*, *err*)
  if $root[path]$.`dir`? **then** $meta \coloneqq root[path]$.`meta`
  else $meta \coloneqq fs[root[path]$.`fid`$]$.`meta`

`posix_writemeta`(*path*, *meta*; *err*)
  if $root[path]$.`dir`? **then** $root[path]$.`meta` $\coloneqq meta$
  else $fs[root[path]$.`fid`$]$.`meta` $\coloneqq meta$;
  $sync \coloneqq false$

Figure 7.4: File Operations of the Component *POSIX* (error handling omitted)

writing meta data such as access rights and access times is performed by the operations `posix_readmeta` and `posix_writemeta`, respectively.

data type `seek-flag` = `SEEK_SET` | `SEEK_CUR` | `SEEK_END`

The invariant proofs, access checking and error handling are described in more detail by Ernst [43, Ch. 7]. Error handling is quite intricate and constitutes a large part of the specification.

7.3 Crash-Safety & Correctness of File Systems

The crash behavior of the component *POSIX* is expressed in terms of synchronized states and an additional crash transition as shown by (posix-crash).

synchronized states (posix-crash)

$sync$

crash

$root' = root \wedge fs' = fs$

recovery

$\texttt{posix\_recover}()\ \{\ fs := root.\texttt{fids} \triangleleft fs, ofh := \emptyset, sync := true\}$

The crash effect in addition to the retraction of several operations, arbitrarily changes *ofh* and *sync* and leaves the directory tree and file store unchanged. Recovery removes all orphaned files by restricting the file store to all files reachable from the directory tree and clears the opened file handles.

Since all operations are atomic the effect on a power cut is that the system retracts several operations, but never across a call to `posix_sync`, and then retries one operation, yielding an alternative trace for the system. If an implementation of a file system satisfies this guarantee, we consider it functionally correct and crash-safe by Def. 7.1.

Definition 7.1 (Functional Correctness & Crash-Safety of File Systems)**.** A file system is *functionally correct* and *quasi-sequentially crash-safe*, if and only if it refines the formal POSIX specification given in this chapter.

The term *quasi-sequential crash-safety* is chosen based on work by Bornholt et al. [18], which defines *sequential crash-safety*. The difference to Def. 7.1 is that sequential crash-safety does not permit the re-execution of one operation, which is necessary for the compositionality result of Ch. 4.

Notice that an explicit account of the effect of write buffering described briefly in Ch. 6 is completely missing from the specification (posix-crash) of the crash behavior on this level of abstraction. The entire effect is captured by the synchronized states and the implicit retraction of operations.

The Flashix file system introduced in Ch. 2 and presented in more detail in the following chapters refines the POSIX component and is therefore functionally correct and quasi-sequentially crash-safe according to Def. 7.1.

7.4 Related Work

Models of POSIX There are a lot of specification of the POSIX or similar interfaces for file systems at different levels of abstraction. Most focus on the aspect of functional correctness and some make strong simplifications or require additional invariants.

A mapping from paths to directories and files is used in [63, 50, 96] as an abstract specification for a file system. An additional invariant is necessary guaranteeing that the prefix of a valid path is also valid. Of the three models hard links are only supported in [96].

In [39] the directory tree is formalizes as a pointer structure where each node stores a parent pointer with an additional invariant that ensures acyclicity.

Heisel [61] uses an algebraic tree to evaluate specification languages.

The POSIX component treats preconditions similar to [63], i.e., if an error occurs then the state remains unchanged.

In [29] an algorithm for path resolution with symbolic links is proven correct.

A more detailed comparison of our model of POSIX can be found in [43].

POSIX & Crashes Bornholt et al. [18] define crash consistency models for POSIX-compliant file systems, based on operations that produce (potentially many) update events. A crash is then expressed by taking a prefix of the update events. The difference between their definition of sequential crash consistency [18, Def. 5] and quasi-sequential crash consistency Def. 7.1 is that the latter allows a re-execution that might produce different events and not just (a reordering of) a prefix. Furthermore, the POSIX model also allows an additional effects of the crash afterwards, i.e., all files are closed after the crash and all orphans removed. Their notion of crash consistency omits the effect on opened and orphaned files. Update events also have a lot of overhead in the specification, because it is necessary to a) specify how the operations act on the state and b) the events and how the events update the state. Crash-aware components basically use the operations as the events. Furthermore, [18] does not support truncation of files.

Effects of a crash on a file system at the level of the POSIX specification is discussed in [11, 26], too. The abstract model of [11] keeps an explicit history back to the most recent flush as a list of higher-order state transformers at a level of abstraction below POSIX. It is proved that the implementation of `sync` correlates to reducing the history to produce a current state. Chen's thesis [26] discusses a specification methodology of write-back caches that are not order-preserving. It is also based on explicit histories and rewriting of histories.

8

Flash Memory: Pages, Blocks, Erasing and Sequential Writes

Summary. The Flashix file system implements the POSIX specification based on flash memory. This chapter describes the specification component *Flash*, which formalizes the assumptions the Flashix file system makes about flash hardware. As a blueprint the Memory Technology Devices (MTD) interface used in Linux is chosen. It is implemented by every flash driver for raw flash devices on Linux. This also ensures that an integration with Linux of the C code, that is derived from the implementation components of Flashix is possible. The component takes the limitations of ONFI-compliant flash hardware, which is an industry standard for flash memory, and its error-prone nature into account.

Publications. The model of flash memory and its invariants are published in [108].

Contents

8.1 Pages and Blocks of Flash Memory

This chapter defines our assumptions about the hardware, captured by the behavior of an abstract interface representing the driver. The interface is based on Linux's Memory Technology Devices (= MTD) interface to flash drivers.

Flash hardware can be categorized into two distinct types [89], which use different types of logic gates. The first flash devices were based on NOR technology, while newer hardware uses NAND gates. NOR flash memory allows for random-access, while NAND flash only supports page-based and sequential access within one block. On the flip side NAND gates are smaller and therefore allow for more storage capacity in the same form factor. The limitations of NAND flash subsume those of NOR flash and we therefore follow the *Open NAND Flash Interface* (= ONFI) standard [4] in order to support both storage technologies.

A NAND flash device is composed of (erase) blocks, each of which contains several pages. A page usually has a capacity of around 512 bytes to 4 KiB. A block consists of approximately 32 to 128 pages. Additionally, there is usually some out-of-band (OOB) data used by the flash driver for error-correction codes in order to compensate for bit flips. Additionally, there is usually one bit to mark a block as bad, which is either set initially by the manufacturer

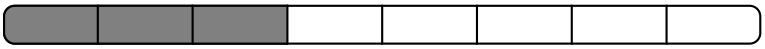

Figure 8.1: A physical erase block of Flash Memory consisting of eight pages. The gray pages were written sequentially. The white pages are still in the erased state and all their bytes contain the value 0xFF.

if some blocks already have errors at the time of fabrication or can be used later by the system to exclude some blocks after they are worn out. Blocks can be read arbitrarily, but writing is only possible *sequentially* within a block and only at the granularity of entire pages. Fig. 8.1 depicts a physical erase block with eight pages, the first three of which are already programmed. The pages of a block can not be overwritten directly. First, the block must be erased in its entirety and only afterwards the file system may perform write operations on the pages of the block again. Initially and after an erase operation, all bits in a block are set to 1. In the figure and throughout this thesis pages in their initial, erased state are depicted in white. Erase is a slow process and we will move this process to a background operation in the erase block manager presented in Ch. 10. After around 10^3 to 10^6 erase cycles, depending on the storage technology used, a block is physically worn out and can not be reused reliably.

These hardware characteristica are modeled by the flash component depicted in Fig. 8.2. In the model flash memory is organized as an array of *physical erase blocks* (= PEBs):[1]

state $pebs$: $\texttt{Array}\langle\texttt{Peb}\rangle$

Each PEB stores a byte-array `data` of fixed length `PEB_SIZE` that is implicitly partitioned into pages of length `PAGE_SIZE`.

$$\textbf{data type}\quad \texttt{Peb} = \texttt{mkpeb}(\texttt{data}\colon \texttt{Array}\langle\texttt{Byte}\rangle, \texttt{written}\colon \mathbb{N}, \texttt{bad}\colon \mathbb{B}) \tag{PEB}$$

A PEB stores a page-aligned counter `written` that tracks the part of the block that contains programmed pages, i.e., only data above `written` is known to be `EMPTY` and can still be programmed before an erase operation is necessary. The constant `EMPTY` denotes the byte value 0xFF, i.e., all bits set. Note that the counter cannot be accessed by software. It is an auxiliary state only used to enforce that pages are written sequentially and never overwritten. PEBs also carry a hardware-supported marker `bad` that is set by the erase block manager of Ch. 10 to prevent future usage of the block.

The model maintains the invariant $\texttt{flash-inv}(pebs)$,

invariant $\texttt{flash-inv}(pebs)$

defined by Equation (flash-inv).

$$\begin{aligned}
\texttt{flash-inv}(pebs) \quad &\leftrightarrow\ \forall p.\, p < \#\, pebs \rightarrow \texttt{peb-inv}(pebs[p]) && \text{(flash-inv)}\\
\texttt{peb-inv}(peb) \quad &\leftrightarrow\ \ \#\, peb.\texttt{data} = \texttt{PEB_SIZE}\\
&\quad \wedge \big(\, \neg\, peb.\texttt{bad} \rightarrow\ \ \texttt{page-aligned}(peb.\texttt{written})\\
&\qquad\qquad \wedge\, \texttt{is-empty}(peb.\texttt{data}, peb.\texttt{written}, \texttt{PEB_SIZE})\,\big)
\end{aligned}$$

It specifies that the counter `written` is a multiple of `PAGE_SIZE`, which is a consequence of the fact that only page-aligned writes are permitted.

$$\texttt{page-aligned}(n) \ \leftrightarrow\ n\ \%\ \texttt{PAGE\_SIZE} = 0$$

[1] The term *physical erase blocks* is chosen to distinguish them from *logical erase blocks*, which are provided as an abstraction by the erase block manager of Ch. 10.

component *Flash*
state $pebs$: $\texttt{Array}\langle\texttt{Peb}\rangle$
initialization
$\texttt{flash\_init}()$ { **choose** $pebs_1$: $\texttt{Array}\langle\texttt{Peb}\rangle$ **with** $\texttt{flash-inv}(pebs_1)$ **in** { $pebs := pebs_1$ } }
invariant
$\texttt{flash-inv}(pebs)$
interface operations
$\texttt{flash\_synchronous\_erase}(p; err)$
pre $p < \#\, pebs \wedge \neg\, pebs[p].\texttt{bad}$
{ $pebs[p] := \texttt{mkpeb}(\texttt{Array}\langle\texttt{Byte}\rangle(\texttt{PEB\_SIZE}, \texttt{EMPTY}), 0, false)$, $err := \texttt{ESUCCESS}$ }
$\vee$ { $\texttt{fail}(; err)$ }
$\texttt{flash\_read}(p, poff, boff, len; buf, isbflip, err)$
pre $p < \#\, pebs \wedge \neg\, pebs[p].\texttt{bad} \wedge poff + len \leq \texttt{PEB\_SIZE}$
$\wedge\ boff + len \leq \#\, buf$
$isbflip := ?$;
{ $buf := \texttt{copy}(pebs[p].\texttt{data}, poff, buf, boff, len)$, $err := \texttt{ESUCCESS}$ }
$\vee$ { $\texttt{fail}(; err)$ }
$\texttt{flash\_write}(p, poff, boff, len, buf; err)$
pre $p < \#\, pebs \wedge \neg\, pebs[p].\texttt{bad} \wedge poff + len \leq \texttt{PEB\_SIZE}$
$\wedge\ boff + len \leq \#\, buf \wedge pebs[p].\texttt{written} \leq poff$
$\wedge\ \texttt{page-aligned}(poff) \wedge \texttt{page-aligned}(len)$
{ $pebs[p] := \texttt{mkpeb}(\texttt{copy}(buf, boff, pebs[p].\texttt{data}, poff, len), poff + len, false)$,
$err := \texttt{ESUCCESS}$ }
$\vee$ { **choose** len_0 **with** $len_0 = 0 \vee len_0 < len \wedge \texttt{page-aligned}(len_0)$ **in**
if $len_0 \neq 0$ **then**
$pebs[p] := \texttt{mkpeb}(\texttt{copy}(buf, boff, pebs[p].\texttt{data}, poff, len_0), poff + len_0, false)$;
$\texttt{fail}(; err)$ }
$\texttt{flash\_is\_bad}(p; isbad, err)$
pre $p < \#\, pebs$
{ $isbad := pebs[p].\texttt{bad}$, $err := \texttt{ESUCCESS}$ } $\vee$ { $\texttt{fail}(; err)$ }
$\texttt{flash\_mark\_bad}(p; err)$
pre $p < \#\, pebs \wedge \neg\, pebs[p].\texttt{bad}$
{ $pebs[p].\texttt{bad} := true$, $err := \texttt{ESUCCESS}$ } $\vee$ { $\texttt{fail}(; err)$ }
$\texttt{flash\_get\_blockcount}(; blockcount)$ { $blockcount := \#\, pebs$ }
$\texttt{flash\_get\_page\_size}(; pagesize)$ { $pagesize := \texttt{PAGE\_SIZE}$ }
$\texttt{flash\_get\_block\_size}(; blocksize)$ { $blocksize := \texttt{PEB\_SIZE}$ }
crash
$pebs' = pebs$

Figure 8.2: Flash Component (Sequential)

The size of a erase block PEB_SIZE is also a multiple of the page size, i.e., the axiom (page-aligned-peb) is assumed.

axiom $\texttt{page-aligned}(\texttt{PEB\_SIZE})$ (page-aligned-peb)

The predicate $\texttt{is-empty}(buf, n, m)$ states that n and m are within the bounds of the array buf and that all bytes in between index n and m contain the value EMPTY and therefore can be programmed by a subsequent writes.

$$\texttt{is-empty}(buf, n, m) \ \leftrightarrow\ n \leq m \leq \#\, buf \wedge \forall i.\, n \leq i < m \rightarrow buf[i] = \texttt{EMPTY}$$

8.2 Operations & Limitations of Flash Memory

The operations of the component *Flash* are also depicted in Fig. 8.2. Note that this component is a specification component and all operations are considered to be atomic, although this is omitted in the figure.

All operations that access an individual physical erase block, given by its index p, have the precondition that the block exists on the device. Except for `flash_is_bad`, which returns whether the given erase block is already marked as bad, all operations additionally have the precondition that the block is not yet worn out.

Most operations may produce a low-level error by non-deterministically choosing between the successful case of the operation and the procedure `fail` defined by (fail).

$$\texttt{fail}(; err) \quad \text{(fail)}$$
$$\textbf{choose } err_0 \textbf{ with } \lfloor err \rfloor \textbf{ in } err \coloneqq err_0$$

This allows for hardware failures, such as being unable to read due to too many uncorrectable bit flips (`EIO`), as well as failures in the driver, e.g., the driver could be unable to allocate sufficient memory to perform an operation (`ENOMEM`).

The operation `flash_synchronous_erase` sets all bits of the physical erase block to 1 and resets the counter `written` to 0, which enables subsequent writes to this physical erase block again.

For reading and writing all offsets and lengths must be in their respective bounds. For `flash_write` the offset into the erase block *poff* and the number of bytes to write *len* must be page-aligned. Reading is not restricted to page-aligned access.

The function $\texttt{copy}(buf_0, i, buf_1, j, n)$, used by the read and write operations, copies n bytes of buffer buf_0 starting at offset i to the buffer buf_1 starting at offset j.

A read may indicate with the *isbflip* output parameter that the error-correction code of the flash driver detected a bit flip. This can be used by the client in order to schedule a wear-leveling cycle for this physical erase block in order to move the data to a more reliable location, before reading from the block completely fails and the data is irrecoverably lost. This is performed by the wear-leveling algorithm of the erase block manager described in more detail in Sec. 10.5.

The operation `flash_write` either writes the entire buffer at the desired offset and increases the counter `written` appropriately, or a hardware failure occurs. In the latter case a low-level error code is returned and only a prefix of the pages is written to the flash device.

The operations `flash_get_page_size` and `flash_get_block_size` ensure that the client components of flash memory are independent of the `PAGE_SIZE` and `PEB_SIZE` constants, which are only an artifact of the verification. For the verification it is sufficient to assume that there are such constants that are unchanged during the lifetime of the component. However, the integration of executable code is easier if the code does not refer to such global constants and instead queries this information during initialization or runtime from the actual flash device.

8.3 Power Failure and Hardware Errors

Access to raw flash devices is usually synchronous and not cached, therefore all states of the model are synchronized and the crash predicate used as a means to express a power failure.

synchronized states	*true*
crash	$pebs' = pebs$

This means that all states are synchronized and no retractions and retries possible. The crash transition itself just states that the physical erase blocks are unaltered, since there are no caches on raw flash.

All operations of the component are introducible (Def. 5.10 on page 54). The operations `flash_is_bad`, `flash_get_blockcount`, `flash_get_page_size` and `flash_get_block_size` do not modify the state. All other operations have the ability to choose the error code `EIO` and leave the state unchanged.

The component *Flash* is the basis for the entire Flashix file system and therefore care should be taken with respect to the assumptions about errors that are implicitly made here. In the following the decisions and their rationale are explained in more detail.

1. Writes of individual pages and block erasure can be viewed as atomic operations.
2. An unsuccessful write of an individual page and an unsuccessful attempt to erase a block do not modify the state.
3. Success of a write operation can be recognized, i.e., an error is not returned by mistake.
4. Conversely, hardware failure can also be detected reliably. In particular, reads that produce garbage can be recognized.
5. An unexpected power failure has no effect on the state of the flash device.

For Assumptions 2 and 3 note the choice if len_0 with the condition

$$len_0 = 0 \vee len_0 < len \wedge \texttt{page-aligned}(len_0)$$

in `flash_write`, i.e., $len = len_0$ is not allowed (if $len \neq 0$) and only a strict, page-aligned prefix of the data is written.

The assumptions are partly justified by the existence of error-correction codes used by the flash driver, which will catch most of these cases in practice.

The motivation behind assumption 1 is that it must be ensured that power failures during these operations must yield an *invalid* physical erase block. However, what exactly invalid means depends on the data structures a client of the component stores on the device. Similarly, assumption 4 requires the client to detect that the data written is actually invalid, when it is read again afterwards. This can be achieved for example by checksums and additional error-correction codes.

Assumptions 3 and 4 ensure that the data structures that are kept in RAM by the file system and the information returned to the user do not arbitrarily diverge from the data stored on the device. Assumption 4 can be relaxed once it is clear what the client stores and what kinds of corrupted data is detectable after reading from flash.

In summary, some of the assumptions are necessary due to the limited expressiveness of the model, but we can relax them later on incrementally, discussed in more detail in Ch. 9 and Ch. 10.

8.4 Related Work

The characteristics of NOR and NAND memory and approaches to deal with them are described in the quite extensive survey [89] and in [56].

Note that the model of flash presented in this chapter does not cover all possible hardware failures that are potentially possible. The model does not cover *read disturbs* [129], where a read can lead to bit flips in adjacent pages. The model assumes that such an error can be detected reliably when reading the other page. A second problem not modeled by the component *Flash* of this chapter are *unstable bits* [132]. If a power failure occurs during writing or erasing then the bits in the block become unstable, and might be read several times afterwards without any errors, but at some point they may suffer from a bit flip. If enough of those bits are flipped then the page or block might become (un)readable. This issue seems to occur only in newer MLC (= Multi-Level Cell) flash devices and cannot be handled solely by the flash driver with error correction codes. However, the erase block manager UBI, which is used in production environments, does not yet handle this error, too.

The models [22, 21, 23] in Z notation of an ONFI-compliant [4] device are conceptually below our model of a driver for flash memory. It would be possible to provide an implementation of our MTD model on top of their hardware model. None of the formal models [22, 75, 38] considers the limitation to sequential writes within an erase block, although non-sequential writes are often not supported by newer ONFI-compliant devices [2, 4].

The flash file system BilbyFs [77, 11, 9] developed at Data61 (formerly NICTA) does not use Linux's MTD interface directly, but builds on top of the erase block manager UBI (see Ch. 10), which already provides error handling, but retains the limitation to sequential, page-aligned writes. Power failures are also not considered formally.

The flash model here does not feature additional out-of-band (OOB) data per page, which is assumed to exist by most Flash Translation Layers (FTLs) [30] and some flash file systems [1, 53]. For the OOB data area the limitations to page-sized and sequential writes only do not apply and individual bits can be programmed. This simplifies the design of FTLs and flash file systems. However, NOR flash devices do not have OOB data and some NAND devices use the whole area for error-correction codes [131].

9 Specification & Verification of (De-)Serialization

Summary. At its core a file system must serialize data structures into a byte-representation and retrieve the contents of the data structure by deserializing the byte-representation. Since this has to happen for various data structures in the following chapters, a generic approach was chosen that integrates well with code generation. It is sometimes necessary to take into account additional constraints on the serialization, such as a fixed or an aligned size, especially when considering the restrictions of flash memory presented in the previous Ch. 8.

Instead of giving a function for data elements that returns the byte-representation, we assume a predicate $\texttt{serialized}\langle\texttt{T}\rangle(t, buf)$,[1] which allows for multiple representations of the same value of type `T` or no representation at all. For example, not all natural numbers might find a representation in 4 bytes of memory and all sequences of the byte representations of the elements of a set might constitute a valid byte representation of the set, preceded by the byte representation for the number of elements. The only two restrictions are axioms (size) and (prefix): The size of the byte-representation of a value t is given by a function $\texttt{serialized-size}\langle\texttt{T}\rangle(t)$.

$$\textbf{axiom}\quad \texttt{serialized}\langle\texttt{T}\rangle(t, buf) \rightarrow \texttt{serialized-size}\langle\texttt{T}\rangle(t) = \#\, buf \tag{size}$$

Furthermore, the prefix of the byte-representation of a value t does not hold any other byte-representations of another value t'.

$$\begin{aligned}\textbf{axiom}\quad & \texttt{serialized}\langle\texttt{T}\rangle(t, buf) \wedge n \leq \#\, buf \wedge \texttt{serialized}\langle\texttt{T}\rangle(t', buf[0..n]) \\ & \rightarrow t' = t \wedge n = \#\, buf\end{aligned} \tag{prefix}$$

For the procedures that perform the actual (de-)serialization the contracts (serialize) and (deserialize) are assumed.

$$\begin{aligned}\textbf{axiom}\quad & off + \texttt{serialized-size}\langle\texttt{T}\rangle(t) \leq \#\, buf \\ & \rightarrow \langle\!|\texttt{serialize}\langle\texttt{T}\rangle(t, off; buf, size, err)|\!\rangle \\ & \quad \big(\ err = \texttt{ESUCCESS} \rightarrow \quad size = \texttt{serialized-size}\langle\texttt{T}\rangle(t) \\ & \qquad\qquad\qquad\qquad \wedge\, \texttt{serialized}\langle\texttt{T}\rangle(t, buf[off..(off + size)])\ \big)\end{aligned} \tag{serialize}$$

The error code `EINVAL` signals that no valid data structure is presented by the buffer. The additional constraint that the buffers length and contents outside of the range off to $off + size$

[1] Technically, in the tool KIV a generic specifications and the mechanism of instantiation of generic specifications are used to derive an instance of the predicate `serialized` for some particular parameter type.

are unchanged are tacitly omitted. Note that for performance reasons operating on a subrange of the buffer is allowed by giving an offset into the buffer. This facilitates allocation of one buffer and iteratively filling the buffer with the byte-representations of several values of the same (or another) type without having to allocate and append several arrays.

axiom $\langle\!|\texttt{deserialize}\langle\texttt{T}\rangle(\mathit{off}, \mathit{buf}; t, \mathit{size}, \mathit{err})|\!\rangle$ (deserialize)

$$\Big(\quad \big(\mathit{err} = \texttt{ESUCCESS} \rightarrow \quad \mathit{off} + \mathit{size} \leq \mathit{buf} \\ \wedge\, \texttt{serialized}\langle\texttt{T}\rangle(t, \mathit{buf}[\mathit{off}..(\mathit{off}+\mathit{size})]) \big) \\ \wedge \big(\mathit{err} = \texttt{EINVAL} \rightarrow \forall n.\ \mathit{off} + n \leq \#\,\mathit{buf} \rightarrow \texttt{is-garbage}\langle\texttt{T}\rangle(\mathit{buf}[\mathit{off}..(\mathit{off}+n)])\big)\Big)$$

A code generator derives an implementation for these procedures and for the function `serialized-size` for every used instance of the parameter types. Having an explicit function `serialized-size` allows for the allocation of a sufficiently large buffer in advance of calling the procedure `serialize`.

Note that we split the assumptions into the predicate and procedures to facilitate code-generation and in a future step the verification of the contracts and axioms on the level of the generated C code by a separate tool such as VeriFast [71] or VCC [32]. These tools only support predicate logic contracts for procedures, instead of the more generic dynamic logic contracts available in KIV. For the verification with KIV an axiom that only involves the two procedures `serialize` and `deserialize` would be sufficient. It would basically state that calling the procedure `deserialize` after serialization with the procedure `serialize` returns the original value, in case both calls are successful.

Now it is also expressible what kind of write errors are acceptable for a value of type `T`, namely those byte sequences that are not a byte-representation of any value of type `T`. This was left open in the hardware model of the previous Ch. 8. This is captured by the predicate $\texttt{is-garbage}\langle\texttt{T}\rangle(\mathit{buf})$.

axiom $\texttt{is-garbage}\langle\texttt{T}\rangle(\mathit{buf}) \leftrightarrow \forall t. \neg\, \texttt{serialized}\langle\texttt{T}\rangle(t, \mathit{buf})$ (garbage)

If the type `T` contains additional measures, such as checksums, then the predicate can be weakened by stating that either no byte-representation or one that is invalid, i.e., contains an invalid checksum, is allowed.

Additional Constraints For a serialization of fixed size, which for example needed to store a subset of the natural numbers into several bytes, we add an additional, nonzero constant for the size and an axiom stating that the size function always returns the constant.

constant $\texttt{serialized-size}\langle\texttt{T}\rangle : \mathbb{N}$ (fixed-size serialization)

axiom $\texttt{serialized-size}\langle\texttt{T}\rangle \neq 0$

axiom $\texttt{serialized-size}\langle\texttt{T}\rangle(t) = \texttt{serialized-size}\langle\texttt{T}\rangle$

Byte-representations with different additional restrictions can be build. For example for flash memory it is sometimes necessary to know that the byte-representation does not contain only `EMPTY` bytes. This can be achieved by signaling an error during serialization when the constraint is violated. Another common constraint is that the byte-representations have a certain alignment. This can be implemented by adding padding bytes until the alignment is reached. The verification of serialization with these constraints and serializations of compound types is rather straightforward and boils down to reasoning about subranges of arrays. One example is presented in the next chapter in Sec. 10.3. Otherwise the details are omitted, only the relevant constraints are stated.

Related Work Related to byte-representations in file systems are specification mechanism for network packets [91] or more general file formats [13] with the ability to generate an implementation for serialization and deserialization procedures.

10 Crash-Safe Erase Block Manager & Wear-Leveling

Summary. This chapter presents one application of the theory of components and crash-safe refinement of Ch. 4, namely the Erase Block Manager (= EBM) of the Flashix file system. As a blueprint for its design and interface the state-of-the-art erase block manager UBI (= Unsorted Block Images) is used. This chapter first captures the interface and expected behavior in an abstract specification and then provides some detail on its implementation and proof of correctness. Of special interest are wear-leveling and asynchronous erasure of blocks, in order to partially deal with the hardware characteristics of flash memory and its performance. The overarching concerns are atomicity with respect to power failures of wear-leveling and the caching of an inverse block mapping caused by the asynchronous erasure of blocks. Another aspect that is considered is whether wear-leveling improves the distribution of erase cycles.

Publications. The specification of the erase block manager is published in [108] and the verification is part of the technical report [109]. This chapter extends both by also considering the quality of wear-leveling.

Contents

10.1 Abstract Specification of an Erase Block Manager

The purpose of the erase block manager is to provide an abstraction over the flash hardware that transparently supports wear-leveling and asynchronous block erasure. The performance of garbage collection and of the commit, both discussed in Ch. 11 are improved by postponing erasure of blocks. A file system that builds on this layer no longer needs to take the degradation

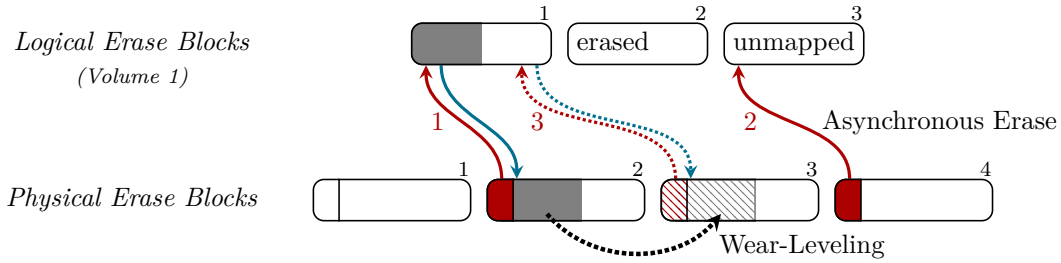

Figure 10.1: Mapping from Logical to Physical Erase Blocks: The **in-RAM mapping** consists of the blue arrows and the **inverse on-flash mapping** is denoted by the red arrows. The labeling of the inverse mapping is its version, i.e., after wear-leveling the new inverse mapping has version **3**, while the previous version is **1**.

of blocks into account. Furthermore, performance problems associated with erasing a block are also dealt with by performing it in the background. Another feature offered by the erase block manager of Flashix and by UBI is atomically exchanging the contents of one block. This is a costly operation and typically only used for the super block of a file system. Atomicity in the case of a super block guarantees that there is no point in time where the super block is invalid or unavailable, which would lead to a complete loss of data after a reboot.

An erase block manager also supports multiple volumes (or partitions) of one flash device. This enables the erase block manager to perform wear-leveling across the entire device, instead of only across the erase blocks assigned to one volume, i.e., wear-leveling and volume management should be coupled on flash hardware in contrast to normal hard disks where volume management is usually performed in a separate layer below all file systems.

The erase block manager does not completely eliminate the characteristics of flash hardware. Errors and partial writes still occur, however, the implementation already performs several attempts to move the data in such cases and to continually perform wear-leveling. Thus, errors surface less often at the level of the client and can basically be treated as fatal errors then, i.e., switching to a read-only mode while still ensuring that a consistent state is reached is sufficient there. Checksums and error-correction codes can and should be used by clients additionally in order to be able to detect more errors or errors that fall outside of the class of errors that can be modeled formally.

In order to be able to move blocks during wear-leveling transparently, an abstraction called *logical erase blocks* (= LEBs) is introduced on top of the physical erase blocks of Ch. 8. The client can only access logical erase blocks, while the implementation maintains a *mapping* from logical to physical erase blocks as depicted by Fig. 10.1. A forward mapping (blue arrows) is stored in RAM for fast access during run-time. On flash only an inverse mapping (red arrows) is persisted, where each physical erase block stores in the first few pages (red area) the logical erase block that is being mapped. Erasing of a block is performed in the background. Therefore, older mappings might still be stored on flash, e.g., the inverse mapping of PEB 4 to LEB 3 in the figure. In case of a power failure, this older mapping is then restored and some logical erase blocks that have already been deallocated reemerge as allocated with arbitrary contents.

In such an implementation wear-leveling copies the data to a new physical erase block and then updates the mapping without disrupting the client. In Fig. 10.1 for example the contents of PEB 2 are moved to PEB 3, including a newer version of the inverse mapping (**dotted red arrow** and **red, shaded part** of PEB 2). Then the forward mapping is adjusted, i.e., the **dotted blue arrow** from LEB 1 to PEB 3 replaces the **solid blue arrow** from LEB 1 to PEB 2. The old version of the inverse mapping (**solid, red arrow** from PEB 2 to LEB 1) is still kept until PEB 2 is actually erased. A valid mapping is only removed once the fist few pages of the PEB are erased.

In order to distinguish between different versions of the *inverse* mapping, version or

sequence numbers are attached to it as shown as a **red labeling** of the arrows of the inverse mapping in Fig. 10.1.

The abstract specification of an erase block manager does not need to model both logical and physical erase blocks and a mapping between them completely, since the distinction is invisible to the client under normal circumstances. However, in the event of a power failure the effects of a cached mapping are observable. It is only necessary to know whether a block is asynchronously erased (unmapped), synchronously erased (erased) or mapped with some contents. A power failure is then expressible by allowing arbitrary contents for LEBs where an asynchronous attempt to erase the corresponding PEB was made.

The remainder of this section introduces the abstract specification of an erase block manager formally, taking the above-mentioned implementation details into account. Sec. 10.2 gives an overview over the implementation. Sec. 10.3 shows how the inverse mapping is encoded and stored. The Sections 10.4 to 10.8 provide details on the state, invariants and operations of the erase block manager's implementation. Afterwards, Sec. 10.9 explains the abstraction relation and the proof of crash-safe refinement. The quality of wear-leveling in terms of an improvement in the distribution of erase cycles by wear-leveling is discussed in Sec. 10.10.

State The complete component *AEBM* is shown in Fig. 10.2 for reference and described in the following in more detail. The state only consists of a finite, partial mapping from volume identifiers $\mathbb{V}$ to volumes. The type of volume identifiers $\mathbb{V}$ is left uninterpreted and can be instantiated with the sort `Byte` for example.

state $avols \colon \mathbb{V} \nrightarrow \texttt{Volume}$

type alias $\texttt{Volume} \equiv \texttt{Array}\langle\texttt{Leb}\rangle$

Each volume has a fixed size and contains logical erase blocks. Similar to PEBs as shown in Equation (PEB) on page 84, LEBs contain data and a counter used to restrict the model to sequential writes only. Bad blocks are hidden by the implementation.

data type $\texttt{Leb} = \texttt{unmapped} \mid \texttt{erased} \mid \texttt{mapped}(\texttt{data}\colon \texttt{Array}\langle\texttt{Byte}\rangle, \texttt{written}\colon \mathbb{N})$

The mapping from logical to physical erase blocks is usually partial in an implementation, i.e., some LEBs might not yet have a corresponding PEB as is for example the case for LEB 2 and 3 in Fig. 10.1. If a LEB is unmapped this means that an asynchronous erase operation was requested by the client and the block might or might not yet have been erased. Asynchronous erasure is a form of caching in the sense that updates to the inverse mapping (**red arrows** in Fig. 10.1) are cached and after a power failure an older version of the mapping might emerge. If a block is erased synchronously this is prohibited.

In order to simplify the presentation we define the selectors `data` and `written` on all constructors of the data type `Leb`.

$$\begin{aligned} leb.\texttt{data} &= \texttt{empty-array}(\texttt{LEB\_SIZE}) && \text{and} \\ leb.\texttt{written} &= 0 && \text{for } leb \in \{\texttt{unmapped}, \texttt{erased}\} \end{aligned}$$

Invariants The invariant (avols-inv) of the abstract specification of an erase block manager just asserts that $\texttt{leb-inv}(leb)$ holds for every logical erase blocks *leb*.

invariant $\texttt{avols-inv}(avols)$ (avols-inv)

where

$$\texttt{avols-inv}(avols) \leftrightarrow \forall \langle v, l\rangle \in avols.\, \texttt{leb-inv}(avols[v][l])$$

component *AEBM*
state $avols\colon \mathbb{V} \nrightarrow \texttt{Array}\langle\texttt{Leb}\rangle$
initialization $\texttt{flash\_init}()\ \{\ avols := \emptyset\ \}$
invariant $\texttt{avols-inv}(avols)$
interface operations

$\texttt{aebm\_create\_volume}(v, size; err)$
pre $v \notin dom(avols)$
$\{\ avols := avols[v, \texttt{Array}\langle\texttt{Leb}\rangle(size, \texttt{erased})],\ err := \texttt{ESUCCESS}\ \}$
$\vee\ \{\ \texttt{fail}(; err)\ \}$

$\texttt{aebm\_get\_volume\_size}(v; volsize)$
pre $v \in dom(avols)$
$\{\ volsize := \#\ avols[v]\ \}$

$\texttt{aebm\_read}(v, l, poff, boff, len; buf, err)$
pre $\langle v, l\rangle \in avols \wedge poff + len \leq \texttt{LEB\_SIZE} \wedge boff + len \leq \#\ buf$
$\{\ buf := \texttt{copy}(avols[v][l].\texttt{data}, poff, buf, boff, len),\ err := \texttt{ESUCCESS}\ \}$
$\vee\ \{\ \texttt{fail}(; err)\ \}$

$\texttt{aebm\_write}(v, l, poff, boff, len, buf; err)$
pre $\langle v, l\rangle \in avols \wedge avols[v][l].\texttt{mapped?} \wedge poff + len \leq \texttt{LEB\_SIZE} \wedge boff + len \leq \#\ buf$
$\wedge\ \texttt{page-aligned}(poff) \wedge \texttt{page-aligned}(len) \wedge avols[v][l].\texttt{written} \leq poff$
$\{\ avols[v][l] := \texttt{mapped}(avols[v][l].\texttt{data}[0..poff] + buf[boff..(boff + len)]$
$+ \texttt{empty-array}(\texttt{LEB\_SIZE} - (poff + len)), poff + len);$
$err := \texttt{ESUCCESS}$
$\vee\ \{$ **choose** len_0 **with** $len_0 = 0 \vee len_0 < len \wedge \texttt{page-aligned}(len_0)$ **in**
if $len_0 \neq 0$ **then**
$avols[v][l] := \texttt{mapped}(avols[v][l].\texttt{data}[0..poff] + buf[boff..(boff + len_0)]$
$+ \texttt{empty-array}(\texttt{LEB\_SIZE} - (poff + len_0)), poff + len_0);$
$\texttt{fail}(; err)\ \}$

$\texttt{aebm\_erase}(v, l; err)$
pre $\langle v, l\rangle \in avols$
$\{\ avols[v][l] := \texttt{erased},\ err := \texttt{ESUCCESS}\ \}$
$\vee\ \{\ avols[v][l] := \texttt{unmapped}; \texttt{fail}(; err)\ \}$

$\texttt{aebm\_unmap}(v, l)$
pre $\langle v, l\rangle \in avols$
$\{\ avols[v][l] := \texttt{unmapped}\ \}$

$\texttt{aebm\_map}(v, l; err)$
pre $\langle v, l\rangle \in avols \wedge \neg\ avols[v][l].\texttt{mapped?}$
$\{\ avols[v][l] := \texttt{mapped}(\texttt{empty-array}(\texttt{LEB\_SIZE}), 0),\ err := \texttt{ESUCCESS}\ \}$
$\vee\ \{\ \texttt{fail}(; err)\ \}$

$\texttt{aebm\_atomic\_change}(v, l, len, buf; err)$
pre $\langle v, l\rangle \in avols \wedge len \leq \#\ buf \wedge len \leq \texttt{LEB\_SIZE} \wedge \texttt{page-aligned}(len)$
$\{\ avols[v][l] := \texttt{mapped}(buf[0..len] + \texttt{empty-array}(\texttt{LEB\_SIZE} - len), len),$
$err := \texttt{ESUCCESS}\ \}$
$\vee\ \{\ \texttt{fail}(; err)\ \}$

$\texttt{aebm\_get\_page\_size}(; pagesize)\ \{\ pagesize := \texttt{PAGE\_SIZE}\ \}$
$\texttt{aebm\_get\_block\_size}(; blocksize)\ \{\ blocksize := \texttt{LEB\_SIZE}\ \}$

crash
$avols \subseteq avols' \wedge \texttt{avols-inv}(avols')$

Figure 10.2: Abstract Specification of an Erase Block Manager

Here and in the following we will use the pair $\langle v, l\rangle$ as the address of a logical erase block and use $\langle v, l\rangle \in avols$ as a shorthand for the formula

$v \in dom(avols) \wedge l < \#\ avols[v]$,

stating that the LEB's address is within bounds of *avols* (and similar data structures).

The invariant on logical erase blocks (leb-inv) is analogous to the invariant on physical erase blocks (flash-inv) on page 84, except that the size of the blocks is smaller, i.e., their size is equal to the constant `LEB_SIZE`, and the bad blocks are hidden from the client by the implementation.

$$\texttt{leb-inv}(leb) \leftrightarrow \quad \#\, leb.\texttt{data} = \texttt{LEB\_SIZE} \wedge \texttt{page-aligned}(leb.\texttt{written}) \\ \wedge\, \texttt{is-empty}(leb.\texttt{data}, leb.\texttt{written}, \texttt{LEB\_SIZE}) \tag{leb-inv}$$

Operations The component provides operations to create new volumes and to get the size of a volume. Creating a volume means that all its logical erase blocks are marked as erased. Reading and writing from LEBs is subject to the same limitations as those imposed by the model of flash hardware of Ch. 8, with the addition that writing is only allowed to mapped blocks. The operations `aebm_erase`, `aebm_unmap` and `aebm_map` are used to affect the mapping of a logical erase block. Unmapping a block does not fail, because it modifies only the in-RAM version of the mapping. Mapping a block sets all its bytes to `EMPTY`. Atomically exchanging the contents of an entire block with `aebm_atomic_change` either successfully writes n bytes of the buffer to the first n bytes of the block and leaves the remainder empty, or does not change the block at all and its old contents are still valid.

Wear-leveling and the actual, asynchronous erasure of unmapped blocks are not observable by the client at this level of abstraction. The correspond to a stuttering step in the specification.

Power Failures All states of the specification are synchronized and therefore no backwards jumps are used to express a power failure. The crash transition of the EBM specification chooses a post-state $avols'$ that satisfies the invariant and retains all data available in the state *avols* just before the power failure.

synchronized states	*true*	(ebm-crash)
crash	$avols \subseteq avols' \wedge \texttt{avols-inv}(avols')$	

More formally, the volumes and their sizes are preserved and all logical erase blocks that were marked as erased or mapped before the crash retain their previous value. Unmapped LEBs, however, may have an arbitrary value afterwards. This captures the fact that the mapping from LEBs to PEBs for asynchronously erased blocks is only cached in RAM in an implementation and the previous mapping and contents might reappear.

$$\begin{aligned} avols \subseteq avols' &\leftrightarrow dom(avols) = dom(avols') &&\wedge\ \forall v \in dom(avols).\ avols[v] \subseteq avols'[v] \\ avol \subseteq avol' &\leftrightarrow \#\, avol = \#\, avol' &&\wedge\ \forall l < \#\, avol.\ avol[l] \subseteq avol'[l] \\ leb \subseteq leb' &\leftrightarrow (leb.\texttt{erased?} \vee leb.\texttt{mapped?} &&\rightarrow leb' = leb) \end{aligned}$$

The volumes and their sizes have to be stored on flash in order to guarantee that the correct volumes and sizes are restored after a power failure, which complicates the implementation.

Crash-Introducibility In order to prove that a client component's operations are atomic with respect to crashes, it is sufficient to prove that the specification is crash-introducible according to Thm. 6 on page 73. For the operation `aebm_write` the crash-introducible run just choses to write the empty prefix. For the operations `aebm_erase` and `aebm_unmap` the case where the LEB is set to unmapped is chosen. The effect of this operation can then be reverted by a power failure according to Equation (ebm-crash). All other operations have a run that does not change the state and are therefore immediately crash-introducible.

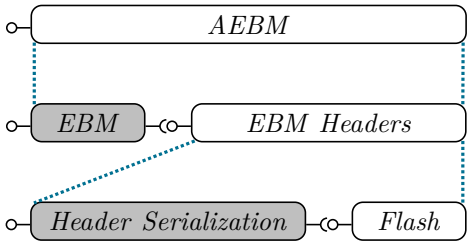

Figure 10.3: Component Structure of the Erase Block Manager

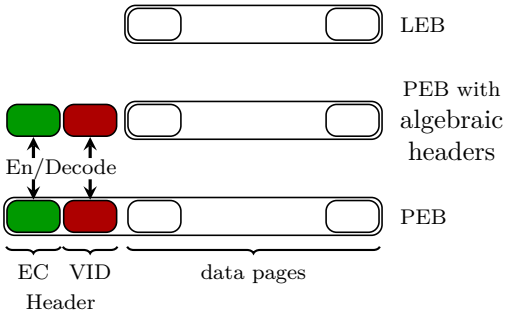

Figure 10.4: Layers of Abstraction: From Physical to Logical Erase Blocks

10.2 Overview over the Implementation

The implementation of the erase block manager is split into two parts as shown in Fig. 10.3: In a first step (component *Header Serialization*) the on-flash data structures for wear-leveling and the inverse mapping depicted in Fig. 10.1 on page 92 are serialized into the first two pages of an erase block. The second step (component *EBM*) then implements the algorithms and data structures for wear-leveling and asynchronous erasure, based on algebraic on-flash data structures. Fig. 10.4 relates the different views on the data of an erase block to each of the levels of abstraction of Fig. 10.3. Bottom-most the physical erase block as seen by the component *Flash* is depicted where all pages are given by a sequence of bytes. In the middle the contents of the first two pages are algebraic data structures instead of raw bytes. In the abstract specification of the previous section only the data pages are observable.

Splitting away a separate component for a data refinement of serialized data structure has in general proven advantageous in the Flashix case study. This measure limits cumbersome reasoning about subranges of arrays and encodings to the subcomponent and does not complicate invariants and the proof of correctness of the "actual" implementation. This pattern is also used several times in subsequent chapters for other on-flash data structures.

The internal structure of the component *EBM* is visualized by Fig. 10.5. Blue nodes represent functional parts. The access of data structures of the component is visualized by arrows. At the core of the component is the *PEB Properties Array*, which stores the current status of each physical erase block, i.e., whether a PEB is currently in use, free, in the process of being erased or already marked as bad. Additionally, the PEB Properties Array stores how often a PEB has already been erased. The latter is called the PEB's *erase count* and it guides the wear-leveling algorithm. The erase count is also stored on flash in the EC-header depicted in Fig. 10.4. For efficient allocation and wear-leveling the array is augmented by two search trees, which store the used and free PEBs, respectively, and are ordered by erase count. The component stores a forward mapping, which is accessed by the interface operations and the wear-leveling algorithm. Additionally, there is a queue of PEBs that need to be erased. The queue is filled by the interface operations with unmapped PEBs and PEBs where a write operation failed. The internal operation responsible for erasure dequeues and erases the PEBs. The physical erase blocks themselves are accessed by each subsystem via the interface of the *EBM Headers*.

10.3 Erase Counter and Volume Identification Headers

Fig. 10.4 on page 96 shows the layout of a PEB. The first two pages are used to store two headers and the remaining pages are data pages usable by the client of the erase block manager.

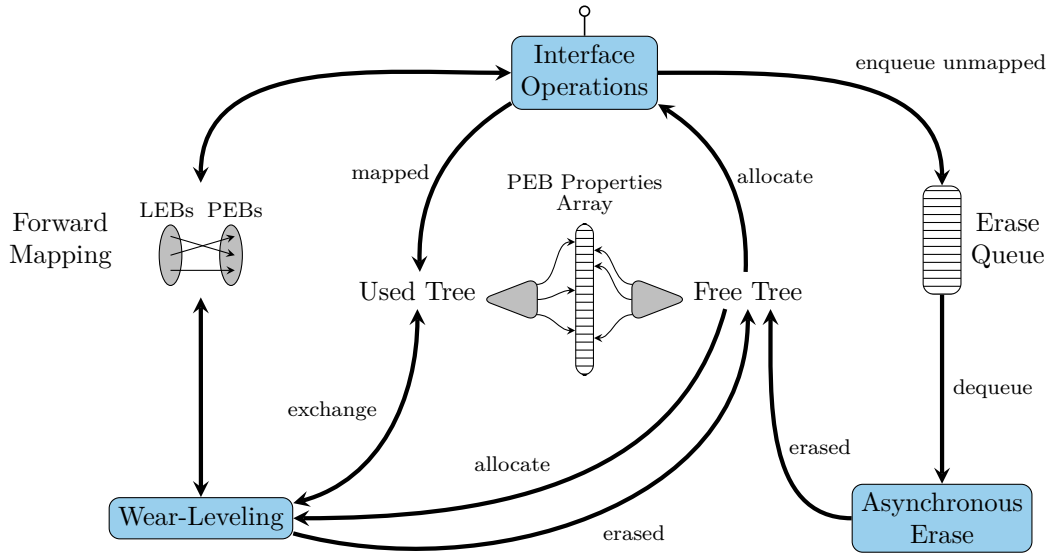

Figure 10.5: Data structures and subsystems of the erase block manager: Functional subsystems are depicted as blue nodes; **bold arrows** denote access to data structures and *thin arrows* denote that both trees are an index for the *PEB Properties Array*.

EC- & VID-Header The first page contains an erase counter associated with the physical erase block (erase count or EC-header). In order to store the data the serialization procedures and abstraction predicates of the previous Ch. 9 are instantiated for the data type `Echdr`.

data type $\texttt{Echdr} = \texttt{echdr}(\texttt{ec}\colon \mathbb{N})$

The erase count is used to allocate appropriate erase blocks and to find suitable erase blocks for wear-leveling.

The second page of allocated PEBs contains the inverse mapping (**red arrows** in Fig. 10.1 on page 92) and is called the volume identifier header (VID-header). The data type `Vidhdr` is used as an algebraic version of the serialized data.

data type $\texttt{Vidhdr} = \texttt{vidhdr}(\texttt{vol}\colon \mathbb{V}, \texttt{leb}\colon \mathbb{N}, \texttt{sqn}\colon \mathbb{N}, \texttt{size}\colon \mathbb{N}, \texttt{checksum}\colon \mathbb{N})$

The VID-header stores the corresponding volume identifier and logical block number of a mapped PEB. Sequence numbers `sqn` distinguish multiple PEBs with equal $\langle \texttt{vol}, \texttt{leb} \rangle$ pairs: The highest sequence number identifies the most recent block for a given inverse mapping. An (optional) size and checksum of the contents of the block are used for atomic block-writes during wear-leveling and is discussed in more detail in Sec. 10.5.

Two headers are necessary, because every non-bad PEB must store its erase counter, but only once a PEB is allocated an inverse mapping is needed. Overwriting a header is not possible due to the limitations of flash hardware. For both headers serialization procedures are used that yield a page-sized buffer that is distinguishable from all bytes set to 0xFF. The serialized data must be different from all bytes set to 0xFF in order to distinguish whether a valid EC-header is present from the situation directly after block erasure or whether the header needs to be written first and whether a PEB is currently mapped and a VID-header exists or whether it is free.

Implementation An excerpt of the component is depicted in Fig. 10.6. As a subcomponent the component *Flash* from Ch. 8 is used. Reading of the headers returns a more general

```
component Header Serialization
  subcomponent Flash                                   (see Fig. 8.2 on page 85)
initialization
  ebm_io_init() { ··· /* write initial EC-headers */ ··· }
interface operations
  ebm_io_read_echeader(p; aehdr, isbflip, err)
    pre p < # pebs ∧ ¬ pebs[p].bad
    let buf = Array⟨Byte⟩(PAGE_SIZE), IsEmpty in
      flash_read(p, 0, 0, PAGE_SIZE; buf; isbflip, err);
      if err = ESUCCESS then
        deserialize-page-sized-nonempty⟨Echdr⟩(0, buf; ehdr, IsEmpty, err);
        if err = ESUCCESS ∧ IsEmpty then
          aehdr := empty
        else if err = ESUCCESS then
          aehdr := mkaechdr(ehdr.ec)
        else if err = EINVAL then
          aehdr := garbage, err := ESUCCESS
  ebm_io_write_echeader(p, aehdr; err)
    pre p < # pebs ∧ ¬ pebs[p].bad ∧ pebs[p].written = 0    { ··· }
  ebm_io_read_vidheader(p; avhdr, isbflip, err)    ···
  ebm_io_write_vidheader(p, avhdr; err)    ···
  ebm_io_read_data(p, poff, boff, len; buf, isbflip, err)    ···
  ebm_io_write_data(p, poff, boff, len, buf; err)    ···
      ⋮
```

Figure 10.6: Component *Header Serialization*: All other operations only forward to the corresponding operation of the subcomponent *Flash* (see Fig. 8.2 on page 85).

value of type `AEchdr` and `AVidhdr`.

```
data type AEchdr = empty | garbage | mkaechdr(ec: ℕ)
data type AVidhdr = empty | garbage
                  | mkavidhdr(vol: 𝕍, leb: ℕ, sqn: ℕ, size: ℕ, checksum: ℕ)
```

This allows the client to distinguish between a situation where a read failed spuriously, which returns an error code, from one where deserialization failed, because the previous write of the header failed and scrambled the bits. The latter returns the special value `garbage` and should lead to the erasure of the erase block. Furthermore, this allows the Flashix file system to deal with more erroneous hardware behavior during writing than explicitly permitted by the hardware model discussed in Ch. 8. The model also allows for writes of bytes that could never be interpreted as a valid header into the first two pages.

Note that the procedures for writing headers and data have as a precondition that the respective part is not yet written, e.g., in the case of the EC-header the precondition states that no bytes of the erase block have been written as shown in Fig. 10.6. The procedures for reading and writing data pages just adds $2 \cdot$ `PAGE_SIZE` to the offset *poff* and calls the subcomponent *Flash*.

Specification The abstract specification *EBM Headers* separates the two, now algebraic headers from the data pages, see also Fig. 10.4 on page 96 for the different views on each level of abstraction.

```
data type APeb = mkapeb(echdr: AEchdr, vidhdr: AVidhdr,
                        data: Array⟨Byte⟩, written: ℕ, bad: 𝔹)
```

component *EBM Headers*
state $apebs$: `Array`⟨`APeb`⟩
initialization
`ebm_io_init`() { ⋯ /* choose $apebs$ with initialized EC-header, empty otherwise */ ⋯ }
invariant
`ebm-io-inv`($apebs$)
interface operations
`ebm_io_read_echeader`(p; $aehdr$, $isbflip$, err)
pre $p < \#\, apebs \wedge \neg\, apebs[p].\texttt{bad}$
$isbflip \coloneqq ?$;
{ $aehdr \coloneqq apebs[p].\texttt{echdr}$; $err \coloneqq \texttt{ESUCCESS}$ } ∨ { `fail`(; err) }
`ebm_io_write_echeader`(p, $aehdr$; err)
pre $p < \#\, apebs \wedge \neg\, apebs[p].\texttt{bad} \wedge apebs[p].\texttt{echdr} = \texttt{empty}$
⋮

Figure 10.7: Component *EBM Headers*

An excerpt of the component is given in Fig. 10.7. Reading the EC-header just takes the value from the abstract PEB and writing (not shown) might write the value `garbage` when it fails. Note again that the precondition for writing must ensure that the header has not yet been written, i.e., that it is still `empty`.

The invariant of the specification is given by the predicate `ebm-io-inv`($apebs$) defined by Equation (ebm-io-inv).

$$\begin{aligned}
&\texttt{ebm-io-inv}(apebs) \leftrightarrow\ \forall p < \#\, apebs.\ \texttt{apeb-inv}(apebs[p]) && \text{(ebm-io-inv)}\\
&\texttt{apeb-inv}(apeb) \leftrightarrow\ \ \#\, apeb.\texttt{data} = \texttt{LEB_SIZE}\\
&\qquad\qquad \wedge \big(\ \neg\, apeb.\texttt{bad} \rightarrow\ \ (apeb.\texttt{echdr.empty?} \rightarrow apeb.\texttt{vidhdr.empty?})\\
&\qquad\qquad\qquad\qquad \wedge (apeb.\texttt{vidhdr.empty?} \rightarrow apeb.\texttt{written} = 0)\\
&\qquad\qquad\qquad\qquad \wedge\, \texttt{page-aligned}(apeb.\texttt{written})\\
&\qquad\qquad\qquad\qquad \wedge\, \texttt{is-empty}(apeb.\texttt{data}, apeb.\texttt{written}, \texttt{LEB_SIZE})\ \big)
\end{aligned}$$

The invariant is obviously quite similar to the invariant for physical and logical erase blocks given by Equation (flash-inv) on page 84 and Equation (leb-inv) on page 95, respectively. The differences are highlighted in **green** and just propagate the restriction that erase blocks need to be written sequentially, i.e., the EC-header is written first while the VID-header and the data pages are still empty. Only then the VID-header may be filled and afterwards data pages may be programmed.

Refinement In order to prove a refinement between the implementation *Header Serialization* from Fig. 10.6 and the specification *EBM Headers* from Fig. 10.7, the abstraction relation (abs-ebmio) is used.

$$\begin{aligned}
&\textbf{abstraction relation}\quad \#\, pebs = \#\, apebs && \text{(abs-ebmio)}\\
&\qquad\qquad \wedge \forall n < \#\, pebs.\ \texttt{abs-peb}(pebs[n], apebs[n])
\end{aligned}$$

The abstraction just relates individual erase blocks by the predicate `abs-peb` defined by the Equation (abs-peb)

$$\begin{aligned}
&\texttt{abs-peb}(peb, apeb) \qquad\qquad (\text{abs-peb})\\
\leftrightarrow\quad &(\ peb.\texttt{bad} \leftrightarrow apeb.\texttt{bad}\ )\\
&\wedge\ (\neg\, peb.\texttt{bad} \rightarrow\ \ peb.\texttt{written} = \texttt{abs-peb-written}(apeb)\\
&\qquad\qquad \wedge\ \texttt{abs-peb-echdr}(peb.\texttt{data}[0..\texttt{PAGE_SIZE}], apeb.\texttt{echdr})\\
&\qquad\qquad \wedge\ \texttt{abs-peb-vidhdr}(peb.\texttt{data}[\texttt{PAGE_SIZE}..2\cdot\texttt{PAGE_SIZE}], apeb.\texttt{vidhdr})\\
&\qquad\qquad \wedge\ apeb.\texttt{data} = peb.\texttt{data}[2\cdot\texttt{PAGE_SIZE}..]\ )
\end{aligned}$$

with

$$\begin{aligned}
&\texttt{abs-peb-written}(apeb)\\
=\ &(apeb.\texttt{echdr.empty?} \supset 0\\
&\qquad\qquad ;\ (apeb.\texttt{vidhdr.empty?} \supset \texttt{PAGE_SIZE}\\
&\qquad\qquad\qquad\qquad ;\ apeb.\texttt{written} + 2\cdot\texttt{PAGE_SIZE}\ )\ )
\end{aligned}$$

and

$$\begin{aligned}
&\texttt{abs-peb-echdr}(buf, \texttt{empty}) &&\leftrightarrow\ buf = \texttt{empty-array}(\texttt{PAGE_SIZE})\\
&\texttt{abs-peb-echdr}(buf, \texttt{mkaechdr}(n)) &&\leftrightarrow\ \texttt{serialized}\langle\texttt{Echdr}\rangle(\texttt{echdr}(n), buf)\\
&\texttt{abs-peb-echdr}(buf, \texttt{garbage}) &&\leftrightarrow\ \texttt{is-garbage}\langle\texttt{Echdr}\rangle(buf)
\end{aligned}$$

and $\texttt{abs-peb-vidhdr}(buf, avhdr)$ defined analogously to $\texttt{abs-peb-echdr}(buf, aehdr)$.

The invariant and refinement proofs are quite easy and just boil down to using the contract for the (de)serialization operations of Ch. 9 and reasoning about subranges of arrays. Note that this is the point of this additional layer of abstraction. Its only purpose is facilitating the verification of the erase block manager presented in the remainder of this chapter by concealing the details of (de)serialization of headers.

10.4 Forward & Inverse Mapping, Reading & Writing

The previous section showed an abstraction of physical erase blocks that separates the two headers needed for the erase block manager from the data stored by its client. The remainder of this chapter presents the RAM state of erase block manager, its operations and verification.

The full state and invariants of the component are depicted in Fig. 10.8 and are introduced incrementally. Note that preconditions of the operations are essentially the same as those of their respective counterpart in the specification of Sec. 10.1, just expressed over a different state space. The preconditions are therefore omitted in the figures that follow. The component *EBM* uses the I/O component *EBM Headers* of the previous section as a subcomponent.

This sections first discusses part of the state and its invariants and shows some of the operations of the component exemplary.

Forward Mapping The forward mapping *vols* (**blue bold arrows** in Fig. 10.1 on page 92) is stored in RAM. It maps each volume identifier $v \in vols$ of a created volume to an array, which is indexed by logical block numbers.

state $vols\colon \mathbb{V} \nrightarrow \texttt{Array}\langle\texttt{PebMapping}\rangle$

data type $\texttt{PebMapping} = \texttt{unmapped} \mid \texttt{mapped}(\texttt{peb}\colon \mathbb{N})$

The value stored is either a physical block number if one has been allocated, or the constant `unmapped` otherwise. If a LEB is unmapped reads return bytes set to `EMPTY` to the client, which corresponds to the initial state of a PEB, after erasure.

PEB Properties Array The second core data structure is the *PEB Properties Array* given by the state variable *ppa*. It is depicted in the center in the overview provided by Fig. 10.5 on page 97. The array stores whether a physical erase block is free, allocated, scheduled for erasure or is already unusable, alongside the PEB's erase counter.

state $ppa\colon \texttt{Array}\langle\texttt{PebProps}\rangle$
data type $\texttt{PebProps} = \texttt{peb-props}(\texttt{ec}\colon \mathbb{N}, \texttt{state}\colon \texttt{PebState})$
data type $\texttt{PebState} = \texttt{FREE} \mid \texttt{USED} \mid \texttt{ERASE} \mid \texttt{BAD}$
invariant $\#\, ppa = \#\, apebs$

The state of a physical erase block stored in the array *ppa* must correspond to the actual state of the block according to the Invariant (ppa-inv).

invariant (ppa-inv)

$$\begin{aligned}\forall p < \#\, ppa.\quad & \big(\ ppa[p].\texttt{state} = \texttt{BAD} \leftrightarrow apebs[p].\texttt{bad}\ \big)\\ \wedge\ & \big(\ ppa[p].\texttt{state} = \texttt{FREE} \rightarrow apebs[p].\texttt{vidhdr} = \texttt{empty}\ \big)\\ \wedge\ & \big(\ ppa[p].\texttt{state} = \texttt{FREE} \vee ppa[p].\texttt{state} = \texttt{USED}\\ & \qquad \rightarrow apebs[p].\texttt{echdr} = \texttt{mkaechdr}(ppa[p].\texttt{ec})\ \big)\\ \wedge\ & \big(\ ppa[p].\texttt{state} = \texttt{USED} \rightarrow \exists v, l.\ \texttt{maximal}(v, l, apebs[p], apebs)\ \big)\end{aligned}$$

The invariant states that a PEB marked as free only has an EC-header, but not yet a VID-header or any data written to it. Furthermore, the erase counter persisted in the header matches the counter stored in the array *ppa*.

Valid, Maximal PEBs and the Inverse Mapping The most important part of Invariant (ppa-inv), however, is that used physical erase blocks are *valid* and *maximal* for some logical erase block $\langle v, l\rangle$. The discussion on what exactly constitutes a valid PEB is postponed to Sec. 10.5 (Equation (valid-peb) on page 106), where atomicity of wear-leveling is considered. For now the following intuition is sufficient: A PEB is valid for the logical erase block $\langle v, l\rangle$ if the PEB has a VID-header that stores $\langle v, l\rangle$ in the respective fields.

A PEB is *maximal* for a LEB $\langle v, l\rangle$ as defined by (maximal-peb) if it is valid and the sequence number (or version number) is the maximum over the sequence numbers of all PEBs valid for LEB $\langle v, l\rangle$.

$$\begin{aligned}& \texttt{maximal}(v, l, apeb, apebs) && \text{(maximal-peb)}\\ \leftrightarrow\ & \texttt{valid}(v, l, apeb)\\ \wedge\ & apeb.\texttt{vidhdr.sqn} = max\big\{\ apebs[p].\texttt{vidhdr.sqn} \ \big|\ p < \#\, apebs \wedge \texttt{valid}(v, l, apebs[p])\ \big\}\end{aligned}$$

The maximum is defined as zero if no valid physical erase block exists, although this is never needed since *apeb* is always part of *apebs*.

In order to guarantee that maximal PEBs are uniquely determined, Invariant (unique-sqns) enforces that PEBs that are valid for a LEB (v, l) have a unique sequence number.

invariant $\mathbb{1}_{\texttt{sqns}(v,l,apebs)}(n) \leq 1$ for all v, l and n (unique-sqns)

The function $\texttt{sqns}\colon \mathbb{V} \times \mathbb{N} \times \texttt{Array}\langle\texttt{APeb}\rangle \rightarrow \texttt{Multiset}\langle\mathbb{N}\rangle$ picks up the sequence numbers and the number of occurrences of all PEBs that are valid for the given LEB. It is defined recursively over the array.

$$\begin{aligned}& \texttt{sqns}(v, l, []) = \emptyset\\ & \texttt{sqns}(v, l, [apeb] + apebs) = (\texttt{valid}(v, l, apeb) \supset \{apeb.\texttt{vidhdr.sqn}\}; \emptyset) \uplus \texttt{sqns}(v, l, apebs)\end{aligned}$$

component *EBM*
  subcomponent *EBM Headers* (see Fig. 10.7 on page 99)
state $vols: \mathbb{V} \nrightarrow \texttt{Array}\langle\texttt{PebMapping}\rangle$, $ppa: \texttt{Array}\langle\texttt{PebProps}\rangle$,
    $eraseq: \texttt{List}\langle\texttt{EraseInfo}\rangle$, $sqnum: \mathbb{N}$, $bflips: \texttt{Set}\langle\mathbb{N}\rangle$, $doWl: \mathbb{B}$
    $volspeb: \mathbb{N}$, $free: \texttt{Set}\langle\texttt{TreeEntry}\rangle$, $used: \texttt{Set}\langle\texttt{TreeEntry}\rangle$
invariant
  $\#\, ppa = \#\, apebs \wedge$ (ppa-inv) $\wedge$ (unique-sqns) (Page 101)
  $\wedge$ (vols-inv) (Page 102)
  $\wedge$ (bflips-inv) $\wedge$ (eraseq-ppa-inv) $\wedge$ (eraseq-no-dups) $\wedge$ (eraseq-mapping) (Page 103)
  $\wedge$ (used-inv) $\wedge$ (free-inv) $\wedge$ (sqnum-inv) (Page 104)
  $\wedge$ (vtbl-inv) (Page 112)
interface operations
  $\texttt{ebm\_read}(v, l, poff, boff, len; buf, err)$
    if $vols[v][l] = \texttt{unmapped}$ **then**
      $buf := \texttt{fill}(buf, \texttt{EMPTY}, boff, len)$, $err := \texttt{ESUCCESS}$
    else let $p = vols[v][l].\texttt{peb}$, $isbflip = false$ **in**
      $\texttt{aebm\_io\_read\_data}(p, poff, boff, len; buf, isbflip, err)$;
      if $err = \texttt{ESUCCESS} \wedge isbflip$ **then** $bflips :+= p$
  $\texttt{ebm\_unmap}(v, l)$
    if $vols[v][l] \neq \texttt{unmapped}$ **then**
      let $p = vols[v][l].\texttt{peb}$ **in**
        $vols[v][l] := \texttt{unmapped}$;
        $used :-= \texttt{tree-entry}(p, ppa[p].\texttt{ec})$, $bflips :-= p$;
        $\texttt{ebm\_asynchronous\_erase}(p, \texttt{Some}(\langle v, l\rangle))$;
auxiliary operations
  $\texttt{ebm\_asynchronous\_erase}(p, lebaddr: \texttt{Option}\langle\texttt{LebAdr}\rangle)$
    $ppa[p].\texttt{state} := \texttt{ERASE}$;
    $eraseq :+= \texttt{erase-info}(p, lebaddr)$;

Figure 10.8: Component *EBM*: State, Invariants and the Operations for Reading and Unmapping

Referring back to Fig. 10.1 on page 92, Invariant (unique-sqns) states that the labels of all **red arrows** targeting one specific LEB are distinct.

The correspondence of the forward and inverse mapping is guaranteed by the Invariant (vols-inv).[1]

invariant $\langle v, l\rangle \in vols \wedge vols[v][l] = \texttt{mapped}(p)$ (vols-inv)
$\leftrightarrow$ $p < \#\, ppa \wedge ppa[p].\texttt{state} = \texttt{USED} \wedge p \neq volspeb$
$\wedge\ apebs[p].\texttt{vidhdr.vol} = v \wedge apebs[p].\texttt{vidhdr.leb} = l$ for all v, l and p

The invariant basically just says that the **red arrows** of the inverse mapping match the **blue arrows** of the forward mapping in Fig. 10.1 on page 92. Together with Invariant (ppa-inv) it follows that only maximal PEBs are actually mapped. This just states that there is a **blue arrow** in Fig. 10.1 on page 92 only if the corresponding inverse **red arrow's** number is maximal, i.e., before wear-leveling the PEB with version 1 is mapped and afterwards the PEB with version 3 *must be* mapped.[2] Note that the invariants do not enforce that all maximal PEBs are mapped, just if a PEB is mapped it must be maximal.

Note that Invariant (vols-inv) implies that the mapping *vols* is injective, i.e., one PEB is not mapped for more than one LEB.

[1] The state variable *volspeb* is explained in Sec. 10.7 and can safely be ignored for the moment.
[2] Assuming that wear-leveling was successful and left the PEB in a state that is valid for the LEB.

Reading an Erase Block Reading from a logical erase block is implemented as shown in Fig. 10.8. If the logical erase block is not mapped, then the buffer is filled with bytes of the value `EMPTY`. Otherwise, the forward mapping is used to determine the number of the mapped physical erase block p. Then, the requested part of the data pages of PEB p are read from the flash device via the component *EBM Headers*. Note that the component *EBM Headers* adds $2 \cdot$ `PAGE_SIZE` to the offset, i.e., this read is past both headers on the actual flash device.

If a correctable bit flip occurs, i.e., the read succeeded with `ESUCCESS` and the flag *isbflip* is set to true, then the physical erase block number is stored in a dedicated set called *bflips*.

state $\mathit{bflips}\colon \texttt{Set}\langle\mathbb{N}\rangle$

These physical erase blocks are preferred for a wear-leveling cycle in Sec. 10.5, since erasing the block and reusing it might alleviate bit flip errors. Since only PEBs currently in use are eligible for wear-leveling, the Invariant (bflips-inv) must hold.

invariant $\mathit{bflips} \subseteq \{\, p \mid p < \#\, \mathit{ppa} \wedge \mathit{ppa}[p].\texttt{state} = \texttt{USED} \,\}$ (bflips-inv)

Unmapping an Erase Block The interface operation that removes the mapping of a logical erase block is also depicted in Fig. 10.8. If the LEB is mapped then the mapping is reset to the constant `unmapped` and the state of the corresponding physical erase block in *ppa* now indicates that it is about to be erased. For performance reasons all PEBs that are scheduled for an erase cycle are additionally kept in the *erase queue*. The queue does not only store the number of the PEB, but it caches also the original mapping $\langle v, l\rangle$ of the PEB (if one existed) in order to be able to implement the synchronous erase operation efficiently.

state $\mathit{eraseq}\colon \texttt{List}\langle\texttt{EraseInfo}\rangle$
data type $\texttt{EraseInfo} = \texttt{erase-info}(\texttt{peb}\colon \mathbb{N}, \texttt{leb}\colon \texttt{Option}\langle\texttt{LebAdr}\rangle)$
type alias $\texttt{LebAdr} \equiv \mathbb{V} \times \mathbb{N}$

Synchronous and asynchronous erases are discussed in Sec. 10.6 in more detail.

The erase queue only caches information already available in the PEB information array and in the VID-header of the PEB itself. Therefore, several invariants that guarantee consistency of this cache are necessary. Invariant (eraseq-ppa-inv) ensures that a PEB has an entry in the erase queue if and only if it has the state `ERASE` in the PEB information array.

invariant (eraseq-ppa-inv)
$$\{\, p \mid \texttt{erase-info}(p, \_) \in \mathit{eraseq} \,\} = \{\, p \mid p < \#\, \mathit{ppa} \wedge \mathit{ppa}[p].\texttt{state} = \texttt{ERASE} \,\}$$

The function $\texttt{pebs}\colon \texttt{List}\langle\texttt{EraseInfo}\rangle \to \texttt{Multiset}\langle\mathbb{N}\rangle$ (omitted here) just calculates the multiset of all `peb` fields of the erase queue and the Invariant (eraseq-no-dups) then states that a PEB has at most one entry in the erase queue. The symbol $\mathbb{1}_M$ denotes the characteristic function of the multiset M.

invariant $\mathbb{1}_{\texttt{pebs}(\mathit{eraseq})}(n) \leq 1$ for all n (eraseq-no-dups)

Invariant (eraseq-mapping) ensures that if a PEB is valid for some LEB and has an entry in the erase queue, then the correct mapping is also stored in the entry.

invariant $e \in \mathit{eraseq} \wedge \texttt{valid}(v, l, \mathit{apebs}[e.\texttt{peb}])$ (eraseq-mapping)
$\to e.\texttt{leb} = \texttt{Some}(\langle v, l\rangle)$ for all e, v and l

The final step of the interface operation `ebm_unmap` removes the pair of PEB number and its erase counter from the *used tree*. In this binary search tree all PEBs currently marked as used have an entry. The tree is sorted by the erase counter. In addition to the used tree

there is also a *free tree*. Both data structures are used for allocation and for wear-leveling, where a PEB with a suitable erase counter needs to be chosen.

state $used\colon \texttt{Set}\langle\texttt{TreeEntry}\rangle, free\colon \texttt{Set}\langle\texttt{TreeEntry}\rangle$
data type $\texttt{TreeEntry} = \texttt{tree-entry}(\texttt{peb}\colon \mathbb{N}, \texttt{ec}\colon \mathbb{N})$

For brevity only the abstraction of the search tree to a set is shown, in the actual model access to the tree is encapsulated in a dedicated subcomponent. The subcomponent is then refined by an implementation that is based on a red-black tree, which provides logarithmic time complexity for all relevant operations.

Invariants (used-inv) and (free-inv) just capture that the right PEB numbers and erase counters are stored in both trees.

invariant

$$used = \left\{ \texttt{tree-entry}(p, ppa[p].\texttt{ec}) \;\middle|\; p < \#\, ppa \wedge ppa[p].\texttt{state} = \texttt{USED} \right\} \quad \text{(used-inv)}$$

$$free = \left\{ \texttt{tree-entry}(p, ppa[p].\texttt{ec}) \;\middle|\; p < \#\, ppa \wedge ppa[p].\texttt{state} = \texttt{FREE} \right\} \quad \text{(free-inv)}$$

Mapping an Erase Block The operation `ebm_map` is shown in Fig. 10.9. It requests a new physical block p from the free tree. The PEB with the minimum erase counter of all free PEBs is chosen as calculated by `min-ec`.

$$\texttt{min-ec}(free) = min\{\, ec \mid \texttt{tree-entry}(\_, ec) \in free \,\}$$

This decision ensures that the best physical erase blocks, according to their erase counter, are reused as soon as possible.

After the allocation of a free PEB, `ebm_map` tries to write the VID-header of the physical erase block. The requested mapping $\langle v, l\rangle$ is written with the sequence number *sqnum*.

state $sqnum\colon \mathbb{N}$

invariant
$$max\left\{ \biguplus_{v,l} \texttt{sqns}(v, l, apebs) \right\} < sqnum \quad \text{(sqnum-inv)}$$

Invariant (sqnum-inv) ensures that the allocated PEB contains the most recent mapping for the logical erase block $\langle v, l\rangle$, because *sqnum* is larger than the sequence number of any valid PEB. Afterwards, *sqnum* is incremented in order to maintain Invariant (sqnum-inv).

If the VID-header is written successfully, the forward mapping *vols* is adjusted accordingly and the PEB p is moved to the used tree. Otherwise, the PEB p is added to the erase queue, because it might contain a corrupted VID-header and therefore can only be safely reused after erasure.

The entire process of trying to write the VID-header can be repeated several times with different PEBs until it succeeds, in order to improve resilience against unpredictable hardware errors.

Instead of returning the error code `ENOSPC` in line ($\star$) in Fig. 10.9, the implementation tries to synchronously erase a PEB from the erase queue and then uses this block as a result of the allocation. This just uses the facilities explained in Sec. 10.6 and is omitted here for brevity.

Writing an Erase Block The implementation of the operation `ebm_write` is also shown in Fig. 10.9. In the normal case the data pages are written successfully to the physical erase block that is mapped the given logical erase block $\langle v, l\rangle$.

However, if the call was not successful, the data pages programmed by previous writes to the PEB—everything up to the offset *poff*— are read. An attempt is made to *atomically*

interface operations

```
ebm_map(v, l; err)
  err := EIO
  let tries = 0, p = 0 in
    while tries ≤ EBM_MAP_RETRIES ∧ err ≠ ESUCCESS do
      tries := tries + 1;
      ebm_allocate_peb(; p, err);
      if err = ESUCCESS then
        aebm_io_write_vidheader(p, mkavidhdr(v, l, sqnum, 0, 0); err);
        sqnum := sqnum + 1;
        if err = ESUCCESS then
          vols[v][l] := mapped(p), ppa[p].state := USED, used :+= tree-entry(p, ppa[p].ec);
        else
          ebm_asynchronous_erase(p, None);
ebm_write(v, l, poff, boff, len, buf; err)
  let p = vols[v][l].peb in
    aebm_io_write_data(p, poff, boff, len, buf; err);
    if err ≠ ESUCCESS then
      let buf_0 = Array⟨Byte⟩(poff + len), isbflip = false in
        aebm_io_read_data(p, 0, 0, poff; buf, isbflip, err);
        if err = ESUCCESS then
          buf_0 := copy(buf, boff, buf_0, poff, len);
          ebm_atomic_change(v, l, poff + len, buf_0; err);
```

auxiliary operations

```
ebm_allocate_peb(; p, err)
  choose p_0 with tree-entry(p_0, min-ec(free)) ∈ free
    p := p_0, free :−= p_0, err := ESUCCESS
  ifnone
    err := ENOSPC                                                    (⋆)
```

Figure 10.9: Component *EBM*: The implementations of `ebm_map` and `ebm_write`: The operation `ebm_map` maps a logical erase block to a free physical erase block allocated via the auxiliary operation `ebm_allocate_peb`. The interface operation `ebm_write` writes to the data pages of a mapped logical erase block. Both interface operations employ different retry mechanism to increase resilience against hardware errors.

replace the contents of the *logical* erase block with the old data appended by the new data. If this fails we end up with the partially written, original PEB. Note that it is crucial here that moving the old data to a new location is done atomically. Otherwise, an interruption in the middle could yield partially written (old) data, but with a newer mapping. This would lead to a loss of data. Fig. 10.10 depicts the different stages the involved PEBs can have. The state that would be reached by the **red arrow** should not occur if the `ebm_change` operation is atomic with respect to errors and power failures.

The facility to move data atomically is also used by the wear-leveling algorithm and is discussed in the next section.

10.5 Atomic LEB Content Exchange & Wear-Leveling

The mechanism that ensures that exchanging the contents of a LEB is possible atomically is to store additional information in the VID-header—specifically a measure of the minimum size of the data in the PEB and a checksum over this data—and only consider a PEB valid if these additional fields match their expected value. This is the purpose of the `size` and

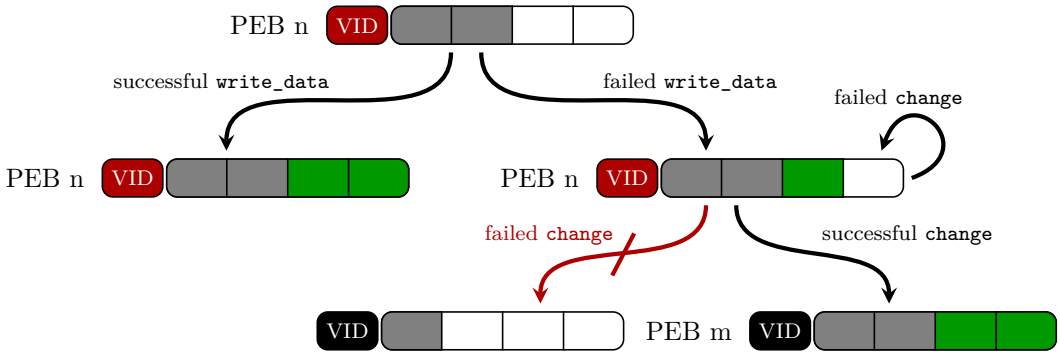

Figure 10.10: The four (legal) stages of writing to a LEB: At each stage only the PEB that is mapped to the LEB is depicted and EC-headers are omitted for clarity. At the top the initially mapped PEB n is shown. If writing is successful PEB n remains mapped with the old and new data. A failed write results in a prefix of the new data being persisted. In this case a retry mechanism tries to move the old and new data to a new PEB m, which is mapped for the LEB if successful. Otherwise, the half-written PEB n remains mapped and a failure is reported to the client. The **black VID-header** has a higher sequence number than the **red VID-header**. Atomicity of `ebm_change` guarantees that the state reached by the **red, crossed arrow** is never observable.

`checksum` fields of the VID-header that were unused and set to zero up until now.

Valid PEBs Formally, we consider the PEB *apeb valid* for some logical erase block $\langle v, l \rangle$ if and only if $\texttt{valid}(v, l, apeb)$ holds as defined by Equation (valid-peb).

$$\begin{aligned} & \texttt{valid}(v, l, apeb) \qquad \text{(valid-peb)} \\ \leftrightarrow \quad & \neg\, apeb.\texttt{bad} \wedge apeb.\texttt{echdr.mkaechdr}? \\ & \wedge \exists sqn, size, chk. \quad apeb.\texttt{vidhdr} = \texttt{mkavidhdr}(v, l, sqn, size, chk) \\ & \qquad \wedge \big(\; size \neq 0 \rightarrow \quad size \leq \texttt{datasize}(apeb.\texttt{data}, \texttt{LEB\_SIZE}) \\ & \qquad\qquad \wedge\, chk = \texttt{checksum}(apeb.\texttt{data}, size) \; \big) \end{aligned}$$

A valid physical erase block is not marked as bad, has valid EC- and VID-headers. The mapping stored in the VID-header corresponds to the LEB $\langle v, l \rangle$. Furthermore, the offset of the last non-`EMPTY` byte in the data pages, which is calculated by $\texttt{datasize}(apeb.\texttt{data}, \texttt{LEB\_SIZE})$, is not less than the `size` field of the VID-header and the checksum over the first `size` bytes matches the `checksum` field of the VID-header.

As a checksum algorithm for example CRC32 could be used. We only demand axiomatically that two buffers buf_0 and buf_1 that contain the same elements up to offset *size* return the same checksum, i.e., $\texttt{checksum}(buf_0, size) = \texttt{checksum}(buf_1, size)$ holds in this case.

Atomically Exchanging LEBs Fig. 10.11 shows the implementation of the procedure `ebm_atomic_change_peb`, which atomically exchanges the contents of the LEB $\langle v, l \rangle$ with the first *len* bytes from buffer *buf* by leveraging Equation (valid-peb). It writes into the data pages of the newly allocated PEB *to* an appropriately prepared VID-header and the contents of the buffer. Note that a page-aligned number of bytes must be written, i.e., the offset of the last non-`EMPTY` byte len_0 is aligned to the next page boundary for writing by passing $\texttt{align}\uparrow(len_0, \texttt{PAGE\_SIZE})$ to the write operation.

In order to understand the error and power failure behavior of this operation better and how it relates to the validity of a PEB as defined by Equation (valid-peb), Fig. 10.12

auxiliary operations

```
ebm_atomic_change_peb(v, l, to, len, buf; err)
  let len_0 = datasize(buf, len), avhdr in
    avhdr := mkavidhdr(v, l, sqnum, len_0, checksum(buf, len_0));
    sqnum := sqnum + 1;
    aebm_io_write_vidheader(to, avhdr; err);
    if err = ESUCCESS ∧ len_0 ≠ 0 then
      aebm_io_write_data(to, 0, 0, align↑(len_0, PAGE_SIZE), buf; err);
    if err = ESUCCESS then {
      if vols[v][l] ≠ unmapped then {                                    (⋆)
        let p_0 = vols[v][l].peb in
          ppa[p_0].state := ERASE, eraseq :+= erase-info(p_0, Some(⟨v, l⟩));
          used :−= tree-entry(p_0, ppa[p_0].ec);
      }
      vols[v][l] := mapped(p), ppa[p].state := USED, used :+= tree-entry(p, ppa[p].ec);
    } else
      ebm_queue_erase(p, Some(⟨v, l⟩)) ;
```

interface operations

```
ebm_atomic_change(v, l, len, buf; err)
  let to = 0 in
    ebm_allocate_peb(; to, err);
    if err = ESUCCESS then
      ebm_atomic_change_peb(v, l, to, len, buf; err);
```

Figure 10.11: Component *EBM*: Atomic LEB Exchange (used by wear-leveling and when writing the superblock in Ch. 12)

visualizes the different stages during and after the operation. Note that here only PEB *to* is shown at each stage.In all stages where PEB *to* is dashed, it is not (yet) valid for the LEB $\langle v, l\rangle$ and the previous PEB that was and is still valid for $\langle v, l\rangle$ is chosen, i.e., the mapping is not updated during the operation and the recovery after a power failure would also chose the previous PEB as discussed in Sec. 10.8 in more detail.

At the top the newly allocated, free PEB *to* is shown. The bold arrows denote state transitions due to a call of an I/O operation. An unsuccessful write to the VID-header leads to an invalid PEB, since either the VID-header is empty or contains garbage. After a successful write of the VID-header, the erase block is still invalid, since the data size and checksum fields of the VID-header do not match the data size of the four empty data pages, which is zero.[3] If the copying is successful, the data size and checksum fields match the contents of the block and the in-memory mapping can be updated accordingly. Otherwise, the new physical erase block is scheduled for erasure and the old PEB is used.

Note that the checksum is only calculated up to the initial data size. Thus, a successive write to the LEB afterwards maintains that the data size and checksum stored in the VID-header match the values calculated from the contents of the data region. Therefore, validity of the PEB is maintained by successive writes.

The operation `ebm_atomic_change` that atomically exchanges the contents of a logical erase block is also shown in Fig. 10.11 and is built around `ebm_atomic_change_peb`. If the operation fails the LEB is unchanged. It only needs to allocate a fresh, free PEB and call `ebm_atomic_change_peb`. In contrast to `ebm_write`, `ebm_atomic_change` is more general and has a more favorable behavior with respect to failures. However, this comes at the price

[3]This assumes that the client did not pass a buffer filled with bytes of the value `EMPTY` to the operation `ebm_atomic_change_peb`. In this case, the PEB is already valid, but obviously also already contains the desired contents.

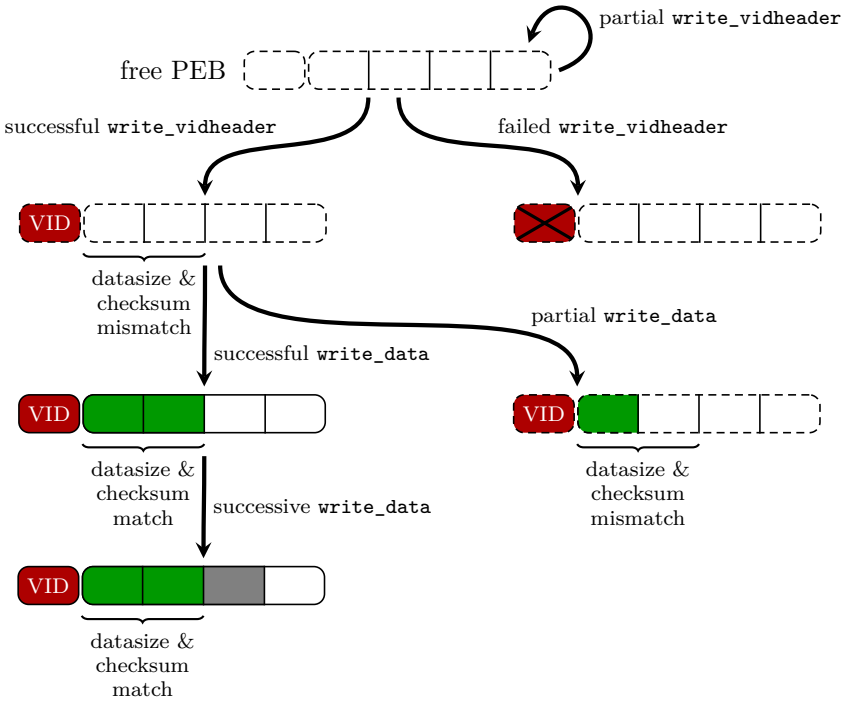

Figure 10.12: States of the new PEB during and after an atomic LEB exchange: The exchange operation should write the first two pages with the **green** data. The dashed states are invalid for the corresponding LEB, either because the VID-header is invalid or the `size` and `checksum` fields of the VID-header do not match the contents of the data pages. In contrast, the solid states are valid. Even after a successive write of the third page to the PEB, it remains valid for the LEB, since the data size only increases and the checksum is unchanged. (EC-headers omitted)

of one additional erasure of a block. Thus, it is only desirable if the additional guarantees are actually required, for example in order to write a new version of the super block during a commit as shown in Ch. 12.

Wear-Leveling The internal operation for wear-leveling is shown in Fig. 10.13. It is triggered by the other operations by setting the state variable $doWl$ to true.

state $doWl \colon \mathbb{B}$

The internal operation additionally performs the task of *scrubbing* erase blocks. Scrubbing moves data from blocks with bit flips to fresh blocks in order to erase the block. Future reuse is usually possible afterwards. Otherwise, the internal operation chooses a used and a free physical erase block of low resp. high wear. Note that the free and used tree are implemented by binary search trees ordered by the erase counter of each entry. Therefore this choice is of logarithmic time complexity. If the difference of the erase counters exceeds a certain threshold, wear-leveling is performed. First, the VID-header and data region of the used PEB are read. Reading the VID-header yields the LEB the PEB is currently mapped for. Afterwards, the data is moved atomically to the new PEB *to* and the forward mapping for the corresponding LEB is updated to point to PEB *to*.

The exact choice of a free PEB for wear-leveling has several reasons. The PEB should have a relatively high erase counter. This is ensured by the maximum of the choice of to_0 in Fig. 10.13 and by the check that its erase counter is larger as that of $from_0$ by at least the threshold. However, the erase counter should not be too high, since we want to delay the use

internal operations

`ebm_wear_leveling`(; *err*, *isScrubbing*)
  guard *doWl*
  let *to*, *from*, *avhdr*, *isbflip*, *buf* = `Array`⟨`Byte`⟩(`LEB_SIZE`) **in**
    `ebm_get_wear_leveling_pebs`(; *to*, *from*; *err*, *isScrubbing*);
    if *err* = `ESUCCESS` **then**
      `aebm_io_read_vidheader`(*from*; *avhdr*, *isbflip*, *err*);
    if *err* = `ESUCCESS` ∧ *avhdr*.`mkavidhdr`? **then**
      `aebm_io_read_data`(*from*, 0, 0, `LEB_SIZE`; *buf*, *isbflip*, *err*);
    if *err* = `ESUCCESS` **then**
      `ebm_atomic_change_peb`(*avhdr*.`vol`, *avhdr*.`leb`, *to*, `LEB_SIZE`, *buf*; *err*);

auxiliary operations

`ebm_get_wear_leveling_pebs`(; *to*, *from*, *err*, *isScrubbing*)
  if *bflips* ≠ ∅ **then**
    ..., *isScrubbing* := *true*
  else
    choose $from_0$ **with** `tree-entry`($from_0$, *min-ec*(*used*)) ∈ *used* **in**
      choose to_0 **with** ∃*ec*. `tree-entry`(to_0, *ec*) ∈ *free*
        ∧ *ec* = *max*{ *ec*′ | `tree-entry`(_, *ec*′) ∈ *free*
          ∧ *ec*′ < *min-ec*(*free*) + 2 · `WL_THRESHOLD` }
      in
        if *ppa*[*from*].`ec` + `WL_THRESHOLD` ≤ *ppa*[*to*].`ec` **then**
          ..., *isScrubbing* := *false*

Figure 10.13: Component *EBM*: Wear-Leveling

of potentially bad blocks as long as possible. Therefore only PEBs with an erase counter that is only higher than the minimum by twice the threshold. Furthermore, the criterion should be algorithmically efficient to check, since the other operations check it when the *free* and the *used* are modified and trigger wear-leveling by setting the flag *doWl* when appropriate.

Note that in order for wear-leveling to work correctly and transparently it is necessary that successive write operations to the mapped PEB are still possible. This is guaranteed, because `ebm_atomic_change_peb` only writes up to the offset

`align`↑(`datasize`(*buf*, `LEB_SIZE`), `PAGE_SIZE`).

However, writes are always sequential within an erase block and therefore always occur after the last non-`EMPTY` byte, i.e., all successive writes must start above this offset and are therefore still permitted.

Preserving PEB Validity This insight is also essential for the verification of the Invariant (ppa-inv) on page 101, which requires preserving the validity of a PEB over successive writes. Invariant (ppa-inv) is the main source of difficulty during the verification of the invariants, exactly because reasoning about the validity of PEBs is quite difficult.

Formally, by the definition of validity (valid-peb) on page 106, it is necessary to prove that (In-)Equalities (10.1) hold for the physical erase block *apeb* that is affected by the write in order to preserve the validity of *apeb* over a successive write.

$$\texttt{datasize}(\texttt{copy}(buf, boff, apeb.\texttt{data}, poff, len), \texttt{LEB\_SIZE}) \geq size \tag{10.1}$$
$$\texttt{checksum}(\texttt{copy}(buf, boff, apeb.\texttt{data}, poff, len), size) = \texttt{checksum}(apeb.\texttt{data}, size)$$

In (10.1) the variable *size* is a shorthand for *apeb*.`vidhdr.size` and all other non-state variables are the input of the operation `ebm_write` from Fig. 10.9 on page 105: The left-hand side denotes the state after a write of *len* bytes to *apeb* beginning at offset *poff*. Fig. 10.14 depicts

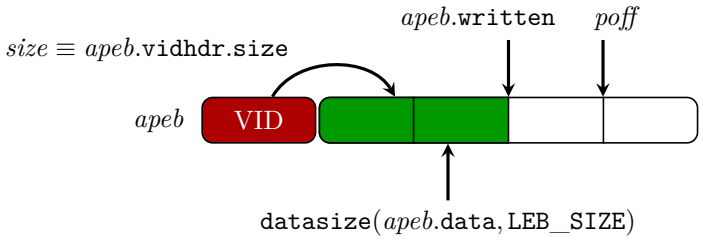

Figure 10.14: Preserving PEB Validity over Successive Writes: The PEB *apeb* is the result of a run of wear-leveling that only wrote the first page and some bytes at the end of the first page are `EMPTY`. Afterwards, a successive write of the second page is performed, which increases $\texttt{datasize}(apeb.\texttt{data}, \texttt{LEB\_SIZE})$ and $apeb.\texttt{written}$ to the depicted values. Another successive write then must start at an offset *poff*, which satisfies the inequalities $apeb.\texttt{vidhdr.size} \leq \texttt{datasize}(apeb.\texttt{data}, \texttt{LEB\_SIZE}) \leq apeb.\texttt{written} \leq poff$.

this situation and the inequalities between the different offsets. Since the PEB was valid before the write $\texttt{datasize}(apeb.\texttt{data}, \texttt{LEB\_SIZE}) \geq size$ is known. The precondition of a write operation implies that $apeb.\texttt{written} \leq poff$ also holds. By Invariant (ebm-io-inv) on page 99 $\texttt{datasize}(apeb.\texttt{data}, \texttt{LEB\_SIZE}) \leq apeb.\texttt{written}$ holds. Taken together this implies $size \leq \texttt{datasize}(apeb.\texttt{data}, \texttt{LEB\_SIZE}) \leq poff$. Thus, the subrange of the PEB the checksum is calculated for is unchanged, i.e.,

$$\texttt{copy}(buf, boff, apeb.\texttt{data}, poff, len)[0..size] = apeb.\texttt{data}[0..size]$$

holds and therefore the checksum itself remains the same. With respect to the first inequality of (10.1), the resulting array can be split into the three parts

$$\texttt{copy}(buf, boff, apeb.\texttt{data}, poff, len) = apeb.\texttt{data}[0..poff] + buf[0..len] + apeb.\texttt{data}[poff+len..]$$

where additionally

$$apeb.\texttt{data}[poff + len..] = \texttt{empty-array}(\texttt{LEB\_SIZE} - (poff + len))$$

is known, because $\texttt{datasize}(apeb.\texttt{data}, \texttt{LEB\_SIZE}) \leq poff + len$. With this the following (in-)equalities imply that (10.1) holds.

$$\begin{aligned}
& \texttt{datasize}(\texttt{copy}(buf, boff, apeb.\texttt{data}, poff, len), \texttt{LEB_SIZE}) \\
= \; & \texttt{datasize}(apeb.\texttt{data}[0..poff] + buf[0..len] + \texttt{empty-array}(\ldots), \texttt{LEB_SIZE}) \\
= \; & \texttt{datasize}(apeb.\texttt{data}[0..poff] + buf[0..len], \texttt{LEB_SIZE}) \\
= \; & \big(\texttt{datasize}(buf, len) = 0 \supset \texttt{datasize}(apeb.\texttt{data}[0..poff], \texttt{LEB_SIZE}) \\
& \qquad\qquad ; \; poff + \texttt{datasize}(buf, len) \big) \\
\geq \; & \texttt{datasize}(apeb.\texttt{data}, \texttt{LEB_SIZE}) \geq size
\end{aligned}$$

This quite tricky reasoning should illustrate that ensuring the (in)validity of a physical erase block is quite complex due to the extensive use of subranges and functions applied to them. Note also that this line of reasoning is pervasive in the entire component, because uniqueness of the sequence numbers (unique-sqns) on page 101 and maximality of the currently held mapping (ppa-inv) on page 101 are also based on validity of a physical erase block.

10.6 Synchronous & Asynchronous Block Erasure

Synchronous erasure of a logical erase block is performed by the interface operation `ebm_erase` shown in Fig. 10.15. The operation first unmaps the corresponding PEB and adds it to the

```
interface operations
  ebm_erase(v, l; err)
    ebm_unmap(v, l);
    ebm_synchronous_erase_leb(v, l; err);
auxiliary operations
  ebm_synchronous_erase_leb(v, l; err)
    ··· // call ebm_synchronous_erase_peb for all blocks in eraseq for the LEB ⟨v, l⟩
  ebm_synchronous_erase_peb(p; err)
    err := EIO;
    let tries = 0 in
      while tries ≤ ERASE_RETRIES ∧ err ≠ ESUCCESS do
        ppa[p].ec := ppa[p].ec + 1, tries := tries + 1;
        aebm_io_synchronous_erase(p; err);
        if err = ESUCCESS then
          aebm_io_write_echeader(p, mkaechdr(ppa[p].ec); err);
      if err = ESUCCESS then
          ppa[p].state := FREE, free :+= tree-entry(p, ppa[p].ec) ;
      else
        aebm_io_mark_bad(p; err)
        if err = ESUCCESS then;
          ppa[p].state := BAD;
  ebm_synchronous_erase_all(; err)
    ··· // call ebm_synchronous_erase_peb for all blocks in eraseq
internal operations
  ebm_erase_worker()
    guard eraseq ≠ []
    let p = eraseq.head.peb, err in
      ebm_synchronous_erase(p; err);
      if err = ESUCCESS then
        eraseq := eraseq.tail;
```

Figure 10.15: Component *EBM*: Synchronous & Asynchronous Erase (preconditions omitted)

erase queue, if a PEB is mapped at all. Afterwards, all PEBs in the erase queue that contain a valid mapping for the LEB $\langle v, l \rangle$ are synchronously erased by `ebm_synchronous_erase_leb`. The operation `ebm_synchronous_erase_peb` is then called for each of those PEBs. It tries to erase the block and write a new EC-header with an increased erase counter several times. If it succeeds the PEB is marked as free and placed into the free tree. Otherwise, an attempt is made to physically mark the block as bad. If the attempt fails, the block remains in the erase queue.[4]

There is also an operation `ebm_synchronous_erase_all` that attempts to erase all blocks in the queue and is called before the file system is unmounted and before a new volume is created, discussed in more detail in the next section.

The background operation for asynchronous erasure is `ebm_erase_worker` and dequeues an entry from the erase queue and then also tries to erase the PEB synchronously via `ebm_synchronous_erase_leb`.

[4] In the actual models in such situations an error message is emitted, such that the user is kept informed of such critical failures.

10.7 Volume Management

In order to support volumes a *volume table* needs to be kept on flash, which stores the existing volumes and their sizes. The state variable *volspeb* records the PEB holding the most recent version of the volume table.

state $volspeb \colon \mathbb{N}$

Updates to the volume table also need to be performed atomically. Therefore, the facilities of Sec. 10.5 for the atomic exchange of the contents of a LEB are reused for the volume table.

Invariant (vtbl-inv) associated with the volume table then just states that PEB *volspeb* is marked as used and valid for the LEB (`VTBL_VOLID`, `VTBL_LEB`), which implied by Invariant (ppa-inv) on page 101. Furthermore, the data pages contain the set `to-vtbl`(*vols*), which is just defined as the set of pairs of volume identifiers and the corresponding volume's size in *vols*.

$$\begin{aligned}\textbf{invariant}\quad & volspeb < \#\, ppa \wedge ppa[volspeb].\texttt{state} = \texttt{USED} \\ & \wedge\ apebs[volspeb].\texttt{vidhdr.vol} = \texttt{VTBL\_VOLID} \\ & \wedge\ apebs[volspeb].\texttt{vidhdr.leb} = \texttt{VTBL\_LEB} \\ & \wedge\ \texttt{serialized}\langle \texttt{Set}\langle \mathbb{V} \times \mathbb{N} \rangle\rangle(\texttt{to-vtbl}(vols), apebs[volspeb].\texttt{data}) \\ & \wedge\ \texttt{VTBL\_VOLID} \notin vols\end{aligned} \qquad \text{(vtbl-inv)}$$

where

$$\texttt{to-vtbl}(vols) = \left\{ \langle v, \#\, vols[v] \rangle \;\middle|\; v \in vols \right\}$$

With respect to the actual serialization, the predicate `serialized` first stores the byte representation of the number of elements, afterwards one possible sequence of those elements and then padding bytes to fill up the space up to `LEB_SIZE`. Thus, the volume table fills up all data pages of one physical erase block.

The volume table only stores the user-accessible volumes and only those are also part of the forward mapping *vols*. Apart from user-accessible volumes, there are also hidden volumes. We currently only use the hidden volume `VTBL_VOLID` to store the volume table itself.[5] Thus, Invariant $\texttt{VTBL\_VOLID} \notin vols$ must be maintained.

A new volume is created as shown in Fig. 10.16. It is first checked that enough space is available in the volume table to hold another entry. Afterwards, a fresh, free PEB is allocated and the contents of the (user-inaccessible) LEB (`VTBL_VOLID`, `VTBL_LEB`) are exchanged *atomically* with the new, serialized volume table. Note again that atomicity is required in order to maintain a consistent and valid volume table.[6] Otherwise, either some or all volumes might be lost if an error or power failure occurs at an intermediate stage of writing. Only if the volume table is written successful, the newly created volume is added to the forward mapping *vols* and all entries are set to `unmapped`.

Note that deleting or resizing a volume can be implemented and verified similarly.

10.8 Initialization, Power Failures & Recovery

Initialization of the component *EBM* writes initial EC-headers with a counter of 0 to all non-bad blocks. In the PEB information array *ppa* all bad blocks are marked accordingly

[5] The Linux implementation of UBI uses additional hidden volumes for a feature termed *Fastmap*, which increases the speed during recovery after the file system is unmounted. The idea essentially is to store a version of the forward mapping *vols* in the first blocks of the device when the file system is unmounted and reconstruct the RAM state from this during normal mounting. However, in the case of a recovery after a power loss, the data is restored from the inverse mapping stored in each block as discussed in Sec. 10.8, since the persisted version might be out of date.

[6] Note that in order for the call to `ebm_atomic_change_peb` to work properly its implementation needs to be adapted appropriately, i.e., the if-statement at line ($\star$) in Fig. 10.11 on page 107 has to handle the case $v = \texttt{VTBL\_VOLID}$ separately and use $p_0 = volspeb$ instead of the PEB found in the forward mapping *vols*.

interface operations

```
ebm_create_volume(v, size; err)
  ebm_synchronous_erase_all(; err);
  if err = ESUCCESS then
    let vtbl = to-vtbl(vols), buf = Array⟨Byte⟩(LEB_SIZE), n, p in
      to-vtbl[v] := size;
      if serialized-size⟨Set⟨𝕍 × ℕ⟩⟩(vtbl) > LEB_SIZE then
        err := ENOSPC
      else
        serialize⟨Set⟨𝕍 × ℕ⟩⟩(vtbl, 0; buf, n, err);
        if err = ESUCCESS then
          ebm_allocate_peb(; p, err)
        if err = ESUCCESS then
          ebm_atomic_change_peb(VTBL_VOLID, VTBL_LEB, p, LEB_SIZE, buf; err);
        if err = ESUCCESS then
          vols[v] := Array⟨Byte⟩(size, unmapped), volspeb := p;
```

Figure 10.16: Component *EBM*: Volume Management

and all other blocks are marked as free except for the PEB *volspeb*. In this PEB an initial volume table that does not contain any volumes is written. All other data structures are initialized accordingly.

The recovery from power loss is by far the largest part of the component *EBM* and constitutes around a third of the code. This difficulty stems from the fact that quite a few data structures have to be rebuilt, since all state variables except for *apebs* are lost in a power failure. The physical erase blocks, however, are assumed to be unchanged by a power failure. Rebuilding the data structures is intricate mainly because checking whether a physical erase block is valid for some LEB as defined by Equation (valid-peb) on page 106 has many cases and steps and needs to be performed for every physical erase block on the device incrementally.

Another contributing factor to the complexity is that an intermediate data structure *ai* (attachment information) is needed during the scanning of the device. There are three reasons for this.

1. The volume table and therefore the existing volumes and their sizes are only known after the scanning is complete and the most recent version of the volume table has been located and read.

2. In order to find the PEB with the highest sequence number for a mapping, it is necessary to cache the highest sequence number for each LEB that is encountered during the scanning process. Then contending PEBs can be discarded or chosen based on a comparison of their sequence number with the current maximum for the LEB

3. Not only the normal volumes need to be restored, but also the hidden volumes, which in our case is only the volume `VTBL_VOLID` with the volume table.

The necessary information for user-accessible and hidden volumes is stored during recovery in the attachment entries mapping *ai*.

$ai : \mathbb{V} \times \mathbb{N} \nrightarrow$ `AttachmentEntry`

data type `AttachmentEntry` = `aientry`(`peb`: ℕ, `sqn`: ℕ)

Fig. 10.17 shows conceptually how the in-memory state is rebuilt from the data structures stored on flash.

First, all physical erase blocks are scanned by the procedure `ebm_scan`, i.e., it is checked whether a PEB is marked as bad and has valid EC- and VID-headers in order to determine

```
recovery
  ebm_recovery(; err)
    let blockcount, valid-ec-mean, invalid-ec-pebs = [] in
      ebm_io_get_blockcount(; blockcount);
      vols := ∅, ppa := Array⟨PebProps⟩(blockcount), eraseq := [], sqnum := 0;
      free := ∅, used := ∅;
      ebm_scan(blockcount; ai, valid-ec-mean, invalid-ec-pebs, err);
      if err = ESUCCESS ∧ ⟨VTBL_VOLID, VTBL_LEB⟩ ∉ ai then
        err := EINVAL;
      else if err = ESUCCESS then let buf = Array⟨Byte⟩(LEB_SIZE), vtbl = ∅ in
        volspeb := ai[⟨VTBL_VOLID, VTBL_LEB⟩].peb
        ebm_io_read_data(volspeb, 0, 0, LEB_SIZE; buf, _, err);
        if err = ESUCCESS then
          deserialize⟨Set⟨V × N⟩⟩(0, buf; vtbl, _, err);
          if err = ESUCCESS then
            ebm_initialize_vol_sizes(vtbl);
            ebm_initialize_vol_mappings(ai);
            ebm_fix_ecs(valid-ec-mean, invalid-ec-pebs);
auxiliary operations
  ebm_scan(blockcount; ai, valid-ec-mean, invalid-ec-pebs, err)
    for i = 0 ... blockcount do
      ...     // Check whether apebs[i] is bad, free, valid or invalid and update
              // ppa[i], ai, eraseq, free and used accordingly
              // Add PEBs with empty or invalid EC-header to invalid-ec-pebs
              // Calculate mean value of all valid erase counters as valid-ec-mean
  ebm_initialize_vols(vtbl, ai);
    ...     // Allocate enough space for each volume in vtbl
            // Move mappings in bounds of vtbl from ai to vols
            // Move all other PEBs referred to by ai to eraseq
  ebm_fix_ecs(valid-ec-mean, invalid-ec-pebs)
    ...     // Set ppa[n].ec to valid-ec-mean for all PEBs n ∈ invalid-ec-pebs
            // with invalid EC-header
```

Figure 10.17: Component *EBM*: Recovery after Normal Reboot or Power Loss

whether it is a valid PEB. If the PEB has a VID-header with a non-zero data size field, then additionally the data pages have to be read and their data size and checksum compared with the expected values as the definition of validity demands. All the data structures are updated during this scanning process with the exception of *vols* and *volspeb*, instead all maximally valid PEBs are stored in the attachment entries map *ai*.

During this process some PEBs might be encountered which are not bad, but nonetheless have an invalid EC-header, because they were in the process of being erased or erasing failed. These PEBs are kept in the list *invalid-ec-pebs* in order to fix their erase counter with some reasonable value later on. As such a value the mean over all erase counters of PEBs with a valid EC-header is chosen and also calculated in the variable *valid-ec-mean* by `ebm_scan`.

Afterwards, it is checked that a volume layout was found during scanning. Mounting fails if no layout is present. Otherwise, the volume table is read and deserialized. For each non-hidden volume identifier a volume of the stored size initialized to `unmapped` is added to *vols* by `ebm_initialize_vols_sizes`. Then the operation `ebm_init_volume_mappings` transfers all mapping information from the intermediate data structure *ai* referring to an existing volume and within its bounds to *vols*.

Finally, for each physical erase block p with an invalid EC-header, its cached erase counter $ppa[p]$.`ec` is set to the mean value over all erase counters by the procedure `ebm_fix_ecs`. This

value is then used after the next erase cycle by `ebm_synchronous_erase_peb` (see Fig. 10.15 on page 111) for the EC-header.

It is crucial for the correctness of the recovery that the in-memory mapping corresponds to the *most recent* (inverse) mapping stored on-disk after each operation, among those PEBs that are *valid*. This is implied by Invariant (vols-inv) on page 102 together with Invariant (ppa-inv) on page 101. To see that this is necessary assume the opposite: There are two PEBs p and p' and both store a mapping for a LEB $\langle v, l\rangle$. In memory $\langle v, l\rangle$ is mapped to p, although p' has the higher sequence number. If the contents of both data regions are initially identical, assume that a write operation is requested by the client on LEB $\langle v, l\rangle$ with non-empty data. Afterwards, the contents of PEB p and p' definitely differ. In the event of a power failure, the subsequent recovery will restore a mapping from $\langle v, l\rangle$ to p'. Reading the mapped LEB $\langle v, l\rangle$ before and after the power-loss will yield different results and therefore data will be lost.

The definition of maximality Def. 10.1 is the key insight necessary to prove the correctness of the recovery mechanism of the component *EBM*.

Definition 10.1 (Maximal State)**.** A state of the component *EBM* is *maximal* if all invariants of the component (see Fig. 10.8 on page 102) are satisfied and additionally Equation (maximal-state) holds, which states that LEBs with pending entries in the erase queue are not marked as unmapped in the forward mapping *vols*.

$$\texttt{lebs}(\mathit{eraseq}) \cap \texttt{unmapped-lebs}(\mathit{vols}) = \emptyset \tag{maximal-state}$$

with

$$\texttt{lebs}(\mathit{eraseq}) = \left\{ \langle v, l\rangle \;\middle|\; \texttt{erase-info}(\_, \texttt{Some}(\langle v, l\rangle)) \in \mathit{eraseq} \right\}$$
$$\texttt{unmapped-lebs}(\mathit{vols}) = \left\{ \langle v, l\rangle \;\middle|\; \langle v, l\rangle \in \mathit{vols} \wedge \mathit{vols}[v][l] = \texttt{unmapped} \right\}$$

The function `lebs` returns all LEBs that have a valid PEB pending in the erase queue. The set of all LEBs that are unmapped in *vols* is returned by the function `unmapped-lebs`.

The most interesting aspect of maximality is summarized in Lem. 10.2.

Lemma 10.2 (Maximal Mapping)**.** *Given two states s and s' of the component EBM over the same device $\mathit{apebs} = \mathit{apebs}'$ that satisfy all invariants of the component (see Fig. 10.8 on page 102) and s' is maximal, then*

$$\mathit{vols} \subseteq \mathit{vols}'$$

holds, where the subset relation between mappings is defined by Equation (mapping-⊆).

$$\begin{aligned} & \mathit{vols} \subseteq \mathit{vols}' && \text{(mapping-}\subseteq\text{)} \\ \leftrightarrow\quad & \mathit{dom}(\mathit{vols}) = \mathit{dom}(\mathit{vols}') \\ & \wedge \forall v \in \mathit{dom}(\mathit{vols}).\quad \#\,\mathit{vols}[v] = \#\,\mathit{vols}'[v] \\ & \qquad\qquad \wedge \forall l < \#\,\mathit{vols}.\ \mathit{vols}[v][l] \neq \mathit{vols}'[v][l] \rightarrow \mathit{vols}[v][l] = \texttt{unmapped} \end{aligned}$$

Proof. Invariants (unique-sqns) on page 101, (ppa-inv) on page 101 and Invariant (vols-inv) on page 102 are crucial for this insight.

The condition of maximality (maximal-state) ensures that if a LEB $\langle v, l\rangle$ is mapped in *vols*, then *vols*′ also provides a mapping. Assume $\mathit{vols}[v][l] = \texttt{mapped}(p)$ for some PEB p, which is marked as used in *ppa*. In the maximal state *ppa*′ the PEB p must be marked either as in use or as scheduled for erasure, according to the Invariant (ppa-inv) (for the states *ppa* and *ppa*′ with the same flash device *apebs*). If the PEB is not in used, according to maximality (maximal-state) there is another PEB for the LEB $\langle v, l\rangle$ that is. This proves that *vols*′ maps more LEBs then *vols*.

According to Invariant (unique-sqns) valid and maximal PEBs are uniquely determined and according to Invariants (ppa-inv) on page 101 and (vols-inv) on page 102 mapped PEBs are valid and maximal. Thus, If a LEB $\langle v, l\rangle$ is mapped in *vols* and in *vols*′, then $\mathit{vols}[v][l] = \mathit{vols}[v][l]$ must holds. □

Since the device is the same for Lem. 10.2, the component *EBM* in its maximal states is able to access the largest possible amount of data. Lem. 10.3 then states that such a state is recovered after a power failure.

Lemma 10.3 (Correctness of Recovery)**.** *The procedure* `ebm_recovery` *terminates and if successful returns a maximal state.*

The proof of Lem. 10.3 only gives a high-level overview and highlights some key steps in the verification.

Proof Outline (Lem. 10.3). The proof is split into four steps. First, a postcondition for `ebm_scan` is established. In a second step it is proven that the most recent version of the volume table is identified correctly. Then the forward mapping and its invariants are restored. In the following variable names with an index zero, e.g., $vols_0$ refer to the state of the component *EBM* before the reboot, variable names without an index refer to the current state.

1. The call to `ebm_scan` recovers a state where all invariants of the component (see Fig. 10.8 on page 102) hold, except for Invariant (vols-inv) on page 102 and Invariant (vtbl-inv) on page 112 since those refer to *vols* and *volspeb*. Instead for the data structure *ai* the postcondition

   $$\texttt{lebs}(eraseq) \subseteq dom(ai) \wedge \text{(scan-ai-post)} \qquad \text{(scan-post)}$$

   is established. The first conjunction ensures that the temporary data structure *ai* contains a mapping for every LEB that is in the erase queue. This means that no PEBs that are valid and maximal are put into the erase queue.

   $$\begin{aligned} & \langle v, l\rangle \in dom(ai) \wedge ai[\langle v, l\rangle] = \texttt{aientry}(p, sqn) && \text{(scan-ai-post)} \\ \leftrightarrow \quad & p < \#\, ppa \wedge ppa[p].\texttt{state} = \texttt{USED} \wedge apebs[p].\texttt{vidhdr.sqn} = sqn \\ & \wedge\, apebs[p].\texttt{vidhdr.vol} = v \wedge apebs[p].\texttt{vidhdr.leb} = l && \text{for all } v, l, p \text{ and } sqn \end{aligned}$$

   The second conjunct (scan-ai-post) ensures that the entries of *ai* are consistent with the inverse mapping and the sequence number stored in the physical erase blocks.

   Establishing Invariant (ppa-inv) on page 101 in `ebm_scan` is quite tricky since checking the validity requires a lot of steps and needs to be matched against its algebraic definition (valid-peb) on page 106.

2. Afterwards, a volume table is read from the PEB

   $$volspeb = ai[\texttt{VTBL\_VOLID}][\texttt{VTBL\_LEB}].\texttt{peb}.$$

   Obviously, it is crucial that the volume table that was in use before the reboot is read. This follows from the fact the $volspeb = volspeb_0$, because according to Invariants (vtbl-inv) on page 112 and (ppa-inv) on page 101 the PEB $volspeb_0$ is valid and maximal for $\langle \texttt{VTBL\_VOLID}, \texttt{VTBL\_LEB}\rangle$ before the reboot.

   The postcondition (scan-ai-post) of the previous step implies that *volspeb* is also a maximal and valid PEB for $\langle \texttt{VTBL\_VOLID}, \texttt{VTBL\_LEB}\rangle$. Physical erase blocks are not altered and maximal and valid PEBs are unambiguous for every LEB according to Invariant (unique-sqns) on page 101, both *volspeb* and $volspeb_0$ must therefore refer to the same block.

3. The call to `ebm_initialize_vol_sizes` first allocates the volumes with their respective sizes are present in the volume table read in the previous step. This step reestablishes Invariant (vtbl-inv) on page 112.

4. Afterwards, `ebm_initialize_vol_mappings` moves all mappings that are within the bounds of existing volumes from the attachment information *ai* to *vols*. Based on the postcondition (scan-ai-post) of the first step, this establishes Invariant (vols-inv) on page 102. All other entries in *ai* are moved to the erase queue. During this process the invariant

   $$\texttt{lebs}(eraseq) \subseteq dom(ai) \cup \texttt{mapped-lebs}(vols),$$

   is maintained, where `mapped-lebs` is defined analogously to `unmapped-lebs`. The invariant holds initially by the first conjunct of the postcondition (scan-post) of the first step. After `ebm_initialize_vol_mappings` completes with $ai = \emptyset$, maximality of the state,

   $$\texttt{lebs}(eraseq) \cap \texttt{unmapped-lebs}(vols) = \emptyset,$$

   follows directly.

The final call to `ebm_fix_ecs` only changes the cached erase counter of PEBs marked as `ERASE`, which does not invalidate any invariants. □

10.9 Verification of Crash-Safe Refinement

The hard part of the verification is actually establishing the invariants of the implementation, since there are several data structures with quite a few consistency requirements between them. In contrast, the abstraction relation has to consider the physical erase blocks *apebs*, the forward mapping *vols* and the erase queue *eraseq* only. The erase queue is used to determine whether a LEB has any valid PEBs. Equation (ebm-abs) shows the abstraction relation.

abstraction relation

$$\begin{aligned} &\texttt{lebs}(eraseq) \cap \texttt{erased-lebs}(avols) = \emptyset \\ &\wedge\ \texttt{abs}(avols, vols, apebs) \end{aligned} \tag{ebm-abs}$$

All LEBs that are marked as `erased` in the specification component are given by the function `erased-lebs`. The invariant therefore states that none of the LEBs marked as `erased` have a valid PEB in the erase queue.

$$\texttt{erased-lebs}(avols) \equiv \left\{ \langle v, l\rangle \;\middle|\; \langle v, l\rangle \in avols \wedge avols[v][l] = \texttt{erased} \right\}$$

The predicate `abs` states that the contents of each logical erase block correspond to the contents of the mapped physical erase block and the number of bytes written to that PEB is not greater than the number of bytes written to the LEB.

$$\begin{aligned}
\texttt{abs}(avols, vols, apebs) \quad &\leftrightarrow \quad dom(avols) = dom(vols) \\
&\quad \wedge \forall v \in vols.\, \texttt{abs-vol}(avols[v], vols[v], apebs) \\
\texttt{abs}(avol, vol, apebs) \quad &\leftrightarrow \quad \#\, avol = \#\, vol \\
&\quad \wedge \forall l < \#\, vol.\, \texttt{abs-leb}(avol[l], vols[l], apebs) \\
\texttt{abs-leb}(leb, ent, apebs) \quad &\leftrightarrow \quad (\ ent = \texttt{unmapped} \leftrightarrow \neg\, leb.\texttt{mapped?}\ ) \\
&\quad \wedge (\ ent.\texttt{mapped?} \rightarrow \quad leb.\texttt{mapped?} \wedge ent.\texttt{peb} < \#\, apebs \\
&\qquad\qquad \wedge\ leb.\texttt{data} = apebs[ent.\texttt{peb}].\texttt{data} \\
&\qquad\qquad \wedge\ apebs[ent.\texttt{peb}].\texttt{written} \leq leb.\texttt{written}\ )
\end{aligned}$$

Note that the abstraction relation is not functional. There are two reasons for this. Firstly, only by observing the specification state *avols* it is known whether a LEB in *avols* is `unmapped` or `erased`. Secondly, the number of bytes written in *avols* might be larger than the corresponding number in *apebs*. The reason is that wear-leveling might decreases this count by writing less bytes to the target PEB than were written to the source PEB, since only non-`EMPTY` bytes are moved.

Theorem 7 (Correctness & Crash-Safety of Erase Block Management & Wear-Leveling)**.** *The component EBM refines the component AEBM using the forward simulation (ebm-abs), and the component Header Serialization refines the component EBM Headers.*

Proof of Thm. 7 for Normal Operations. The second part is already proven in Sec. 10.3.

The proof strategy to maintain the *first conjunct* of the abstraction relation (ebm-abs) is basically to strengthen the invariant theorems of each of the operation such that the postcondition states that $\texttt{lebs}(eraseq')$ of the post-state $eraseq'$ can be derived from the pre-state $eraseq$ in one of the following ways:

- Either the operation might add a LEB $\langle v, l\rangle$ to the erase queue or leave the erase queue unchanged and then

  $$\texttt{lebs}(eraseq') \subseteq \texttt{lebs}(eraseq) \cup \{\langle v, l\rangle\}$$

  holds. This is the case for the operations `ebm_unmap`, `ebm_unmap`, `ebm_write` (due to the retry mechanism), `ebm_change` and `ebm_wear_leveling`. However, for each of these operations the corresponding LEB in *avols* is no longer marked as `erased` afterwards and is therefore not in the set $\texttt{erased-lebs}(avols)$.

- Otherwise, the operation erases one or several PEBs. For `ebm_erase_worker` and the unsuccessful case of `ebm_erase` it is sufficient to prove $\texttt{lebs}(eraseq') \subseteq \texttt{lebs}(eraseq)$. In case of success of `ebm_erase`,

  $$\texttt{lebs}(eraseq') \subseteq \texttt{lebs}(eraseq) \setminus \{\langle v, l\rangle\}$$

  holds. And for `ebm_create_volume` it is proven that $\texttt{lebs}(eraseq') = \emptyset$ and thus that all LEBs of the new volume can be marked as `erased` in *avols*.

For the *second conjunct* of the abstraction relation (ebm-abs), observe that modification of *avols* and *vols* usually occur at some specific LEB $\langle v, l\rangle$, and it is therefore useful to define a version $\texttt{abs}_{v,l}(avols, vols, apebs)$ of the abstraction predicate `abs` that *excludes* this LEB. Then Equations (unfold-leb-a) and (unfold-leb-c) can be used to unfold the modified LEB and then locally prove `abs-leb` for the modified LEB.

$$\begin{aligned} &\texttt{abs}(avols[v \mapsto avols[v][l \mapsto leb]], vols, apebs) && \text{(unfold-leb-a)}\\ \leftrightarrow\ &\texttt{abs}_{v,l}(avols, vols, apebs) \wedge \texttt{abs-leb}(leb, vols[v][l], apebs) \end{aligned}$$

$$\begin{aligned} &\texttt{abs}(avols, vols[v \mapsto avols[v][l \mapsto ent]], apebs) && \text{(unfold-leb-c)}\\ \leftrightarrow\ &\texttt{abs}_{v,l}(avols, vols, apebs) \wedge \texttt{abs-leb}(avols[v][l], ent, apebs) \end{aligned}$$

Modification to the LEB $\langle v, l\rangle$ in *avols* and *vols* then trivially sustain the predicate $\texttt{abs}_{v,l}$.

Writes to the physical erase blocks *apebs* are a bit more difficult, because such a write can fall into two categories.

- A set of PEBs S unreachable by the mapping *vols* can be modified freely as shown by equation (mod-unreach). This is for example the case for blocks that are being erased or are newly allocated and not yet mapped.

  $$\texttt{abs}_{v,l}(avols, vols, apebs) \leftrightarrow \texttt{abs}_{v,l}(avols, vols, apebs') \qquad \text{(mod-unreach)}$$
  $$\text{if} \quad apebs =_{\texttt{mapped-pebs}(vols)} apebs'$$
  $$\text{where } \texttt{mapped-pebs}(vols) = \left\{\, p \;\middle|\; \langle v, l\rangle \in vols \wedge vols[v][l] = \texttt{mapped}(p) \,\right\}$$

  Note that usually the operations do not just modify one PEB, instead the retry mechanisms employed in many parts of the component *EBM* could lead to multiple modified PEBs. This is a complicating factor for the verification, however, such implementation measures are necessary for a robust implementation.

- The PEB that is mapped for the accessed LEB $\langle v, l\rangle$ is altered. This requires that the mapping *vols* is injective as an additional precondition. Otherwise, some other LEB might refer to the same PEB and the changes are non-local. Fortunately the mapping is kept injective by Invariant (vols-inv) on page 102, and Equation (mod-at) can be employed.

$$\texttt{abs}_{v,l}(avols, vols, apebs[p, \_]) \leftrightarrow \texttt{abs}_{v,l}(avols, vols, apebs) \quad \text{(mod-at)}$$
$$\text{if } vols \text{ is injective and } vols[v][l] = \texttt{mapped}(p)$$

  In this case the modification at PEB $vols[v][l].\texttt{peb}$ needs to expressed additionally.

With the equations (unfold-leb-a), (unfold-leb-c), (mod-unreach) and (mod-at) it is essentially possible to show that the abstraction relation (ebm-abs) is maintained over all operations, based on theorems about the operations that limit the changes to the state variables *apebs*, *vols* and *eraseq* sufficiently.

Proving that the outputs are the same usually involves unfolding the predicate `abs-leb` for the required LEB. □

Proof of Thm. 7 for Recovery. Unprimed and primed state variables refer to the state before and after the power failure and recovery, respectively. According to Lem. 10.3 the state of component *EBM* is maximal after the recovery. The state on the abstract level can be chosen with the restriction given by the crash specification (ebm-crash) on page 95 of the component *AEBM*. The new state $avols'$ is chosen based on the previous state *avols* and the maximal state of the component *EBM* after recovery as

$$avols' \equiv \texttt{absf}(avols, apebs, vols', eraseq')$$

where `absf` returns the same volumes and volume sizes as *avols* and $vols'$ (and *vols*) and for each LEB $\langle v, l\rangle$ in bounds, the value

$$\begin{aligned} vols'[v][l] = \texttt{unmapped} \supset \big(\langle v, l\rangle \in eraseq' \supset \texttt{unmapped}; \texttt{erased}\big) \quad \text{(absf)} \\ ; \ \texttt{mapped}\big(apebs[vols'[v][l].\texttt{peb}].\texttt{data}, \\ avols[v][l].\texttt{mapped}? \supset avols[v][l].\texttt{written} \\ ; \ apebs[vols'[v][l].\texttt{peb}].\texttt{written}\big) \end{aligned}$$

is used as $\texttt{absf}(avols, apebs, vols', eraseq')[v][l]$. Basically, it is attempted to mark as many LEBs as possible as `erased` and to reuse the number of bytes written to the LEB for mapped LEBs if it existed. The proof then has to establish the following facts:

- crash specification (ebm-crash) on page 95:

  1. $avols \subseteq \texttt{absf}(avols, apebs, vols', eraseq')$
     All LEBs that are mapped in *avols* are also mapped in *vols*. Since $vols \subseteq vols'$ by Lem. 10.2 and maximality, the same PEB is used in *vols* and $vols'$. Therefore, each LEB in the left-hand side of the inequality is mapped in the right-hand side and derives its data from the same PEB. If a LEB $\langle v, l\rangle$ is `erased` in *avols*, then $vols[v][l] = \texttt{unmapped}$ and $\langle v, l\rangle \notin eraseq$ by the abstraction relation before the crash. It follows that there is no PEB valid for LEB $\langle v, l\rangle$ in *apebs*. Therefore, $vols'[v][l] = \texttt{unmapped}$ and by maximality of the state after recovery Equation (maximal-state) on page 115, it follows that $\langle v, l\rangle \notin eraseq'$. Thus, the right-hand side also evaluates to `erased` for the LEB.
  2. $\texttt{avols-inv}(\texttt{absf}(avols, apebs, vols', eraseq'))$
     This essentially follows from $\texttt{avols-inv}(avols)$ and $\texttt{ebm-io-inv}(apebs)$ and the observation that *avols* and $vols'$ have the same volumes and volume sizes.

- abstraction relation (ebm-abs):

  3. $\texttt{lebs}(\mathit{eraseq}') \cap \texttt{erased-lebs}(\texttt{absf}(\mathit{avols}, \mathit{apebs}, \mathit{vols}', \mathit{eraseq}')) = \emptyset$
     This follows from the construction of `absf` (absf), since only those LEBs are `erased` that do not have an entry in the erase queue eraseq'.
  4. $\texttt{abs}(\texttt{absf}(\mathit{avols}, \mathit{apebs}, \mathit{vols}', \mathit{eraseq}'), \mathit{vols}', \mathit{apebs})$
     The data of each mapped LEB is related by construction of `absf`. The field `written` is either related by construction of `absf` or by the abstraction relation before the power failure.

Together this ensures that a run of the recovery of component *EBM* has an abstract run that propagates the abstraction relation. □

10.10 Quality of Wear-Leveling

This section discussed what kind of guarantees about the *quality* of the wear-leveling algorithm can be given. This is especially important since flash hardware is notoriously unreliable and error-prone.

The goal of wear-leveling is that blocks are worn out evenly. The wear of a block is usually measured by the number of erase cycles that it went through. One measure for the evenness of the wear on a flash device is, how far the erase counters deviate from the maximum erase counter. In the following we will show that wear-leveling decreases the deviation from the maximum erase counter.

The function $\texttt{ecs}(\mathit{ppa})\colon \texttt{Array}\langle\texttt{PebProps}\rangle \to \texttt{Multiset}\langle\mathbb{N}\rangle$ maps the wear-leveling array to the multiset of erase counters and is defined recursively over the array by the equations

$$\texttt{ecs}([]) = \emptyset$$
$$\texttt{ecs}([\mathit{wle}] + \mathit{ppa}) = \{\mathit{wle}.\texttt{ec}\} \uplus \texttt{ecs}(\mathit{ppa})$$

The measure for the distribution of the erase counters is $\Delta\, \mathit{ppa}$ and defined as the distance or deviation of every erase counter to the maximum over all erase counters, weighted by the number of its occurrences.

Definition 10.4 (Better Distribution)**.** A distribution of *ppa* is *better* than the distribution of ppa' if and only if $\Delta\, \mathit{ppa} < \Delta\, \mathit{ppa}'$, where Δ is defined as

$$\Delta\, \mathit{ppa} = \sum_n \left(\mathbb{1}_{\texttt{ecs}(\mathit{ppa})}(n) \cdot \left(\mathit{max}(\texttt{ecs}(\mathit{ppa})) - n \right) \right)$$

The following theorem then shows that wear-leveling improves the distribution.

Theorem 8 (Quality of Wear-Leveling)**.** *A successful run of wear-leveling leads to a better distribution of erase counters or more precisely* $\Delta\, \mathit{ppa}' < \Delta\, \mathit{ppa}$ *where* ppa' *denotes the state after a successful execution of* `ebm_wear_leveling_worker` *with isScrubbing set to false (Fig. 10.13 on page 109) and after the previously used PEB has been synchronously erased.*[7]

Proof. Wear-leveling itself does not change the $\mathit{ppa}[p].\texttt{ec}$ field for any PEB p. However, it choses a PEB *from* that satisfies $\mathit{ppa}[\mathit{from}].\texttt{ec} + \texttt{WL\_THRESHOLD} \leq \mathit{ppa}[\mathit{to}].\texttt{ec}$ for some other PEB *to* in `ebm_get_wear_leveling_pebs` in Fig. 10.13 on page 109. Thus, PEB *from* is not one of the PEBs with the maximum erase counter, assuming (axiomatically) $\texttt{WL\_THRESHOLD} \neq 0$ holds.

The synchronous erase afterwards by the operation `ebm_synchronous_erase_peb` in Fig. 10.15 on page 111 increases $\mathit{ppa}[\mathit{from}].\texttt{ec}$ by `ERASE_RETRIES` at most. Assuming (again

[7] In the actual model wear-leveling synchronously erases the PEB at the end, since we are already in a background operation and have spare time. However, this is omitted from Fig. 10.13 on page 109 for simplicity.

axiomatically) that the inequality ERASE_RETRIES $\leq$ WL_THRESHOLD holds, this update of $ppa[from]$.ec does not alter $max(\texttt{ecs}(ppa))$ in Def. 10.4 of Δ. Furthermore, $max(\texttt{ecs}(ppa)) - ppa[from].\texttt{ec}$ decreases by the number of tries to erase and thus $\Delta\, ppa$ decreases by the same amount. □

Note that the distribution of the current wear $\Delta\, ppa$ is expressed over the RAM data structure ppa. Thus, it might not correspond to the number of actual erases of the block. The erase block manager might experience a power failure multiple times after erasing a block but before writing the erase counter header. In these cases recovery cannot recover an erase counter for these blocks and uses the *average* erase counter over all blocks with a valid erase counter as discussed in Sec. 10.8. This problem, however, is inherent to any wear-leveling algorithm, such as e.g. the algorithm used by UBI, that stores the erase counter of a block in that particular block. It is not deemed critical since power failures are quite rare and therefore the erase counters stored in RAM do not deviate significantly from the actual number of erases.

10.11 Related Work

UBI As a blueprint for the design of the component *EBM* the Linux implementation of a erase block manager called UBI [55] was taken. This ensures that the retry mechanisms and the general handling of hardware failures are realistic and validated in principle by the UBI implementation. Furthermore, this would facilitate other file systems, e.g. BilbyFs or UBIFS, to run on top of the erase block manager presented in this chapter.

Finally, it was possible to transfer some of the knowledge of the verification into the Linux implementation, i.e., the verification presented in this chapter led to the discovery of one bug in UBI in the procedure that corresponds to `ebm_write` (see Fig. 10.9 on page 105).[8] In the implementation of UBI the retry mechanism did not use the facility to atomically exchange the contents of a LEB by storing the amount of data and a checksum in the header. Instead a normal VID-header with a higher sequence number was written. This situation is exactly the one reached by the **red, crossed arrow** depicted in Fig. 10.10 on page 106. In the rare case of a power failure during this retry mechanism this could lead to data loss and potentially to the corruption of the entire file system if a PEB that stores data structures of the UBIFS file system is affected.

One improvement over the wear-leveling algorithm presented in this chapter is [24], which implements a protection mechanism that avoids moving data unnecessarily. This is accomplished by protecting recently wear-leveled and reused PEBs from being chosen for a wear-leveling cycle just because they have a low number of erases. However, the verification conducted in this chapter carry over easily to this setting, since basically only the choice of blocks for wear-leveling are changed, based on some additional data structure.

Flash Translation Layers Flash Translation Layers (FTLs) [30] and some FFSs [1, 53] similarly store information about the state of a page or block in out-of-band (OOB) data, which allows programming of individual bits. This simplifies the recovery from power failures during wear-leveling, since it is possible to set a validity bit after copying the data. However, NOR flash devices do not have OOB data and some NAND devices use the whole area for error-correction codes [131]. Therefore, our EBM implementation is more generic. FTLs that support an operation similar to `unmap` (see "trim" command in Section 7.10 in [68], [79] clarifies the semantics) also have the problem that pages re-emerge after a power failure. Note that FTLs that either 1) update the mapping before copying the actual data during wear-leveling or 2) assume failures that write a valid mapping but invalid data simultaneously have to deal with re-emerging blocks or pages. In the Flashix erase block manager the inverse

[8] The corresponding thread in on the MTD mailing list can be found at http://lists.infradead.org/pipermail/linux-mtd/2016-June/068022.html.

mapping must be updated first because it is stored in the second page and the underlying model of flash hardware enforces that pages are written sequentially. A good overview over the techniques for FTLs is [89].

Formal Models The block manager in the Alloy models [75, 76] maps logical to physical pages and has a similar task as the component *EBM*. However, storing and updating an on-disk mapping is not treated. Power failures are only considered during writing of a sequence of pages. Their specification of power failures and recovery is intertwined and uses auxiliary variables for the status of a pages. It is not immediately clear, how one would disentangle the specification in a real implementation.

In the refinement-based approach [38, 36] with Event-B, it is assumed that bookkeeping information is stored in every page, i.e., a page knows the version of the file it belongs to and the offset within the file. Updating the contents of one page is atomic. If two pages store the same inverse mapping after a power failure during wear-leveling, its contents are identical and choosing either suffices. However, this approach uses more memory for the mapping and requires reading every page of the flash device during startup in order to rebuild the mapping.

The BilbyFs file system [77, 11, 9] is build on top of the interface of UBI, which is identical to the interface of the erase block manager discussed in this chapter. Thus, it would be possible to use the EBM of Flashix as a basis for BilbyFs.

11

Verified Journaling & Garbage Collection

Summary. This chapter shows how the abstract interface of a file system (Component *AFS*) is refined by a journaling flash file system with an index for efficient access. A transactional journal allows the file system to deal with power failures by recording different versions of a file system object and replaying the most recent changes to the file system after a reboot. In order to find the most recent version of a file system object an index is used. Obsolete versions of file system objects are cleaned up by garbage collection. A specific form of a journaling file system, called a log-structured file system, is chosen to deal with the restriction to sequential writes within erase blocks, which is an artifact of the abstraction provided by the erase block manager of Ch. 10. The index is implemented by a wandering B^+-tree and integrated into the commit, recovery and garbage collection schemes.

Publications. The transactional journal and the interface of the persistence layer are published in [46]. Publication [107] shows that the verification can be simplified by using the component semantics of Ch. 4 with retractions and retry to specify the behavior of a power failure. Sec. 11.2 is also based on previous work by Schierl [121] and by Ernst [43, Ch. 9].

Contents

Note that parts of this chapter, namely the component *FFSC* in Sec. 11.2, are based on previous work by Schierl et al. [121], which paved the way for the Flashix file system, and by Ernst [43, Ch. 9]. The concepts, however, need to be repeated here in order to understand the inner workings of the Flashix file system and the contribution of this and the next chapter.

11.1 Overview

The component *FFSC* (= Flash File System Core) implements the interface of a file system as defined by the component *AFS* of Ch. 6 with the help of the subcomponent *Index&Journal* as depicted in Fig. 11.1. The path lookup with access checks and the segmentation of files is already performed by the surrounding component *VFS*.

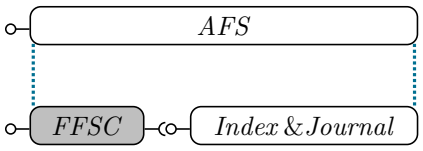

Figure 11.1: Structure of the Component *FFSC*

The interface of the component *AFS* exposes operations to access three different file system objects: *inodes* are identified by an *inode number* and store information about files and directories, *dentries* describe the name and target inode of a directory entry and *pages*. A page is just a segment of a file, i.e., an array of bytes.

The *FFSC* component therefore has to locate the current version of such a file system object and if modifications are requested, write a new version of the file system object *out-of-place*, i.e., to a new location. Thus, already at this stage in the component hierarchy the limitations of flash hardware—the necessity to perform out-of-place updates—play an important role.

However, this is not the only reason why several versions of a file system object are kept. In file system design, the concept of journaling is quite common, file systems such as ext3fs [130] and NTFS support it. The *journal* stores the sequence of modifications made to file system objects. On a recovery it is then possible to replay the journal and only apply those modifications that were completed. The *FFSC* component groups modifications of multiple file system objects accessed by the same operation in a *transaction* in order to guarantee atomicity of the operation even under power failures. File creation for example writes a new version of the parent directory inode, the directory entry and the file inode grouped into one transaction.

Flashix is not only a journaling file system, but it is *log-structured*. In a log-structured file system [116] the entries in the journal fulfill two functions: They have to record modifications to the file system objects such as deletion of file system objects or truncation of files as well as store the actual contents of the file system objects. For a journaling file system this is not required. There it is sufficient to only record modifications to the file system objects in the journal. The actual contents of the objects could be stored separately.

The view on the storage medium in the component *FFSC* is *unstructured*, i.e., it is not yet divided into several blocks, and access to the storage medium and the index is encapsulated in the subcomponent *Index&Journal*. The implementation *Transactions* of this subcomponent introduced a block structure on the storage device and provides facilities to perform atomic transactions and internally garbage collects obsolete versions of file system objects based on a block structured view of the flash device. Garbage collection is integrated with the index in order to be able to distinguish obsolete from live objects. Fig. 11.2 shows the structure of the implementation. In a second refinement step, namely in component *B^+-tree* in the figure, the index is implemented by a wandering B^+-tree.

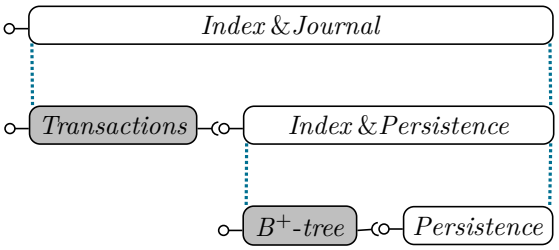

Figure 11.2: Structure of the Components *Transactions* and *B^+-tree*

Sec. 11.2 discusses the state and operations of the component *FFSC* and its interaction with its subcomponent *Index&Journal*. Sections 11.3 to 11.6 focus on different aspects of the component *Transactions* and its correctness. Sec. 11.7 explains the integration of the

index into the commit and recovery mechanism and garbage collection of the index.

11.2 Core Concepts of a Flash File System

RAM Index, Node Store & Log An entry in the journal is called a *node*. Some node types contain actual data and some are only used to store deletion and truncations.

The *FFSC* component locates the current version of a file system object with an index that maps *keys* to *addresses* of the underlying flash medium. Each type of key corresponds to one of the three file system object types. Inode keys identify a file or directory. Data keys identify a page of a file by its inode number and the offset of the page in the file. Dentry keys characterize an edge in the file system tree, i.e., store the inode number of the parent directory and the name of the directory entry.

data type $\texttt{NodeKey} = \texttt{inodekey}(\texttt{ino}\colon \mathbb{N}) \mid \texttt{datakey}(\texttt{ino}\colon \mathbb{N}, \texttt{part}\colon \mathbb{N}) \mid \texttt{dentrykey}(\texttt{ino}\colon \mathbb{N}, \texttt{name}\colon \texttt{String})$

The RAM index *rindex* maps these keys to the address of the node that contains the most recent version of the corresponding file system object. The structure of addresses is discussed in Sec. 11.4 in more detail.

state $\mathit{rindex}\colon \texttt{NodeKey} \rightharpoonup \texttt{NodeAddress}$

Nodes serve the dual purpose of storing the actual data and of recording modifications such as deletion and creation. Additionally, they store the key of the file system object they contain explicitly. This facilitates recovery from a power failure and garbage collection. During recovery the node is replayed by updating the version of the file system object referred in the RAM index. Garbage collection itself also reads the nodes and queries whether an object is still live by performing a lookup of the key in the RAM index. Inode, dentry and data nodes correspond to file system objects with the exception that a dentry node with an `ino` field of 0 signifies deletion of the dentry. Inodes store the meta data, such as access rights, and the number of entries in a directory. For a file the inode store the size and number of hard links to the file. A truncation node does not correspond to a file system object, instead it only records the removal of several pages and a change in the file's size. In addition to the key, the truncation node stores the new file size only.

data type $\texttt{Node} = \texttt{inode-node}(\texttt{key}\colon \texttt{NodeKey}, \texttt{meta}\colon \texttt{metadata}, \texttt{info}\colon \texttt{InodeInfo}) \mid \texttt{dentry-node}(\texttt{key}\colon \texttt{NodeKey}, \texttt{ino}\colon \mathbb{N}) \mid \texttt{data-node}(\texttt{key}\colon \texttt{NodeKey}, \texttt{data}\colon \texttt{Array}\langle\texttt{Byte}\rangle) \mid \texttt{truncation-node}(\texttt{key}\colon \texttt{NodeKey}, \texttt{size}\colon \mathbb{N})$

where

data type $\texttt{InodeInfo} = \texttt{directory}(\texttt{nentries}\colon \mathbb{N}) \mid \texttt{file}(\texttt{nlink}\colon \mathbb{N}, \texttt{size}\colon \mathbb{N})$

Nodes are stored in the node store *ns*, which maps addresses to nodes. The log records the sequence of modifications by storing the addresses of nodes that caused these changes.

state $\mathit{ns}\colon \texttt{NodeAddress} \rightharpoonup \texttt{Node}, \quad \mathit{log}\colon \texttt{List}\langle\texttt{NodeAddress}\rangle$

Flash Index & Commit The log and node store are persistent. In contrast, updates to the RAM index are performed in main memory only. After a power failure therefore the RAM index is lost and needs to be recovered, essentially by replaying all modifications recorded in the nodes of the log in order. More formally, this idea is expressed by Invariant (replay-log-tentative-1)

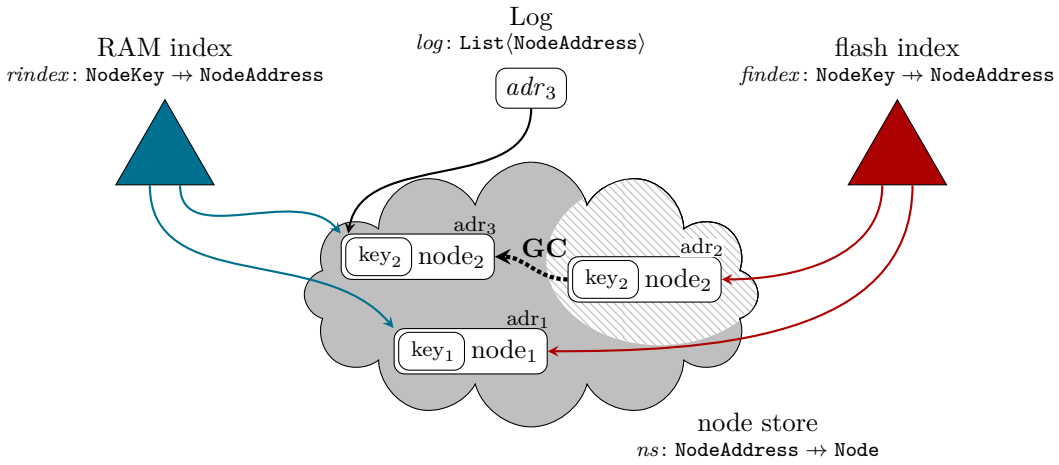

Figure 11.3: Overview over the Concepts of the Flash File System Core: The node store contains the nodes that correspond to current and old version of file system objects. The current version of a file system object is referenced by the RAM index. During a commit the RAM index is persisted to flash. The difference is stored as the log, which needs to be replayed in sequence in order to derive the RAM index from the flash index during recovery. Garbage collection moves live nodes and updates the RAM index afterwards, e.g., node_2 is have been moved to adr_3. Afterwards, nodes no longer referenced by the log or RAM index may be removed, indicated by the shaded part of the node store, potentially leading to dangling references in the flash index.

and refined in the following. Starting from an empty RAM index the changes recorded in the log are applied in order, i.e., for each address $adr \in log$ the RAM index is updated to $adr \mapsto ns[adr].\texttt{key}$. The final result must be equal to the current RAM index.

$$rindex = \texttt{replay-log}(log, ns) \qquad \text{(replay-log-tentative-1)}$$

Replaying all modifications of the file system starting from its initial state is quite expensive and requires all old versions of file system objects to be kept. Therefore, only the modifications since the last *commit* are recorded in the log. During a commit the current version of the RAM index *rindex* is persisted to flash memory and the log is emptied. The flash index *findex* is then the starting point for the recovery operation after a power failure or normal reboot. The RAM index is therefore one of the *commit data structures* referred to in Ch. 6.

state $findex$: `NodeKey` ⇸ `NodeAddress`

With the flash index the above invariant is refined to Invariant (replay-log-tentative-2), but need to be refined further at the end of this section.

$$rindex = \texttt{replay-log}(findex, log, ns) \qquad \text{(replay-log-tentative-2)}$$

Garbage Collection In order to remove old versions of file system objects, garbage collection is employed. Garbage collection copies some nodes to new addresses, updates the RAM index accordingly and then deallocates nodes from the node store. Note that this already captures implementation details, i.e., the implementation of garbage collection selects an entire erase block, moves all live objects and afterwards triggers an asynchronous erasure of the block.

Fig. 11.3 depicts the above-mentioned part of the state of the component *FFSC* and of the component *Index&Journal*. The node store is depicted as a cloud and individual nodes

contain the nodes they are referenced under in the index. The RAM and flash index and the log refer to several of the nodes by their address. The figure also shows the situation before and after a garbage collection run. Initially, the nodes at addresses adr_1 and adr_2 are referenced by the RAM index and by the flash index and the log is empty, i.e., the initial situation is the result of a commit. Garbage collection moves node_2 to address adr_3 and updates the the RAM index so that key_2 now refers to adr_3, resulting in the **blue arrows** in the figure. It also adds adr_3 to the log. Then garbage collection can remove the shaded part of the node store, which includes the old version of the node under adr_2.

In the event of a power failure the RAM index can now be restored by loading the flash index (**red arrows**) and replaying the one log entry adr_3, which updates the RAM index by replacing the entry $\text{key}_2 \mapsto \text{adr}_2$ by $\text{key}_2 \mapsto \text{adr}_3$. In general one node can lead to multiple updates in the RAM index, e.g., truncation nodes need to delete several pages of a file from the index. Note that the flash index might contain references to deallocated nodes, such as adr_2 in the figure, and only after the entire replay is complete the index is consistent with the node store.

Invariants The model has several invariants, which are described in more detail in Schierl et al. [121] and Ernst [43, Ch. 9]. They basically state that the key and node types match, i.e., an inode node contains an inode key and inode keys in the index only refer to inode nodes. In order to implement path lookup it is necessary that both end points of a dentry key in the index are also part of the index. Thus, truncation nodes are never referred to by any index, but only by the log. Furthermore, valid inode numbers are always nonzero, and therefore using 0 to indicate deletion of dentries is permissible. The crucial Invariants (log-cons) and (gc-cons) states that nodes referred to by the RAM index and log are allocated and that the mapping from keys to addresses stored in the RAM index corresponds to the mapping stored in the node itself.

invariant $\texttt{log-cons}(\mathit{log}, \mathit{ns}) \wedge \texttt{gc-cons}(\mathit{rindex}, \mathit{ns})$

defined by

$$\texttt{log-cons}(\mathit{log}, \mathit{ns}) \leftrightarrow \mathit{log} \subseteq \mathit{dom}(\mathit{ns}) \qquad \text{(log-cons)}$$

$$\texttt{gc-cons}(\mathit{rindex}, \mathit{ns}) \leftrightarrow \forall \mathit{key} \in \mathit{rindex}.\ \ \mathit{rindex}[\mathit{key}] \in \mathit{dom}(\mathit{ns}) \wedge \mathit{ns}[\mathit{rindex}[\mathit{key}]].\texttt{key} = \mathit{key} \qquad \text{(gc-cons)}$$

Essentially, the node store contains an inverse mapping from addresses to keys. This is similar to the situation in the erase block manager of Ch. 10, where every erase block stores a candidate mapping and the most recent mapping is distinguished by the highest sequence number among the valid erase blocks. Here the most recent mapping is either already stored in the flash index or is stored in the log and is replayed on top of the flash index.

Invariant (gc-cons) implies that the RAM index is injective. Note that (gc-cons) does not hold for the flash index *findex*, because the flash index might contain dangling addresses as shown by Fig. 11.3 where adr_2 is already garbage collected but still referenced by the flash index.

Index & Journal Subcomponent The component *FFSC* itself only has a set of inode numbers *rorphans* that correspond to the orphaned file of the component *POSIX* (see Ch. 7) as its state. The node store, log and flash index are naturally stored on the flash device and the RAM index is implemented as a *wandering* B^+*-tree* [59]. It is tightly integrated with the flash index, because the index is loaded lazily from the flash device and only modified subtrees of the RAM index are stored during commit. The entire state is therefore part of the *Index&Journal* subcomponent depicted in Fig. 11.4.

However, not all of the invariants carry over to the subcomponent, because the recovery of the RAM index from the flash index and the log is performed by the component *FFSC*.

component *Index&Journal*
state ns: `NodeAddress` $\nrightarrow$ `Node`, log: `List`$\langle$`NodeAddress`$\rangle$
$rindex, findex$: `NodeKey` $\nrightarrow$ `NodeAddress`, $forphans$: `Set`$\langle\mathbb{N}\rangle$, $synced$: $\mathbb{B}$
initialization
`index_journal_init`($volsize$; err)
$ns := \emptyset$, $rindex := \emptyset$, $log := []$, $findex := \emptyset$, $forphans := \emptyset$, $synced := true$
invariant
`inj`($rindex$) $\wedge$ `inj`($findex$) $\wedge$ `log−cons`(log, ns)
interface operations
`index_journal_contains`(key; $exists, err$)
$exists := key \in rindex$;
`index_journal_lookup`(key; adr, err) **pre** $key \in rindex$
$adr := rindex[key]$
`index_journal_update`(key, adr) **pre** $adr \notin ran(rindex) \wedge adr \in ns$
$rindex[key] := adr$
`index_journal_remove`(key)
$rindex :-= key$;
`index_journal_truncate`($ino, size, err$)
$rindex := rindex \setminus \{$ `datakey`$(ino', part) \mid ino' = ino \wedge size \leq part \cdot$ `VFS_PAGE_SIZE` $\}$;
`index_journal_dentries`(ino; $dentries, err$)
$dentries := \{\, name \mid$ `dentrykey`$(ino, name) \in rindex \,\}$;
`index_journal_get`(adr; $node, err$) **pre** $adr \in ns$
$node := ns[adr]$;
`index_journal_transaction`$_i$($node_1, \ldots, node_i$; $adr_1, \ldots, adr_i, err$) for $i \in \{1, \ldots, 5\}$
choose $adr'_1, \ldots adr'_i$ **with** `disjoint`$(adr'_1, \ldots, adr'_i) \wedge$ `fresh`$(adr'_1, \ldots, adr'_i, ns)$ **in**
$adr_1 := adr'_1, \ldots, adr_i := adr'_i$;
$ns := ns[adr_1, node_1] \cdots [adr_i, node_i]$;
$log := log + adr_1 + \ldots + adr_i$;
$synced := false$;
`index_journal_sync`($; err$)
$\{$ $synced := true; err :=$ `ESUCCESS` $\}$ $\vee$ $\{$ `fail`$(; err)$ $\}$
`index_journal_commit`($rorphans$; err)
pre $ran(rindex) \subseteq dom(ns)$
choose $adrset$ **with** $adrset \cap ran(rindex) = \emptyset$ **in**
$findex := rindex$, $forphans := rorphans$, $ns := ns \setminus adrset$, $log := []$, $synced := true$
synchronized states
$synced$
crash
$ns' = ns \wedge log' = log \wedge findex' = findex \wedge forphans' = forphans$
recovery
`index_journal_recover`($; log', rorphans, err$)
$rindex := findex$, $rorphans := forphans$, $log' := log$, $synced := true$

Figure 11.4: Interface operations of the Component *Index&Journal*: Error-Handling and Garbage Collection are omitted. In all erroneous cases the state is left unchanged.

The RAM index only satisfies the invariants of the flash index after the recovery of the subcomponent *Index&Journal*. Specifically, Invariant (gc-cons) does not hold for the flash index. It is only known that the flash index and the RAM index are injective and the Invariant (log-cons) is satisfied by the subcomponent.

invariant `inj`($rindex$) $\wedge$ `inj`($findex$) $\wedge$ `log−cons`(log, ns)

interface operations

```
ffsc_create(meta, parentino, dentry; err)
  choose adr, adr_1, adr_2, adr_3, newino, key_1, key_2, key_4, node_1, node_2, node_3 in
    key_1 := inodekey(parentino);
    key_2 := dentrykey(parentino, dentry.name);
    index_journal_new_ino(; newino);
    key_3 := inodekey(newino);
    index_journal_lookup(key_1, adr; err);
    if err = ESUCCESS then
      index_journal_get(adr; node_1, err);
      if err = ESUCCESS then
        node_1.info.size :+= 1;
        node_2 := dentry-node(key_2, newino);
        node_3 := inode-node(key_3, meta, file(1, 0));
        index_journal_transaction_3(node_1, node_2, node_3; adr_1, adr_2, adr_3, err);
        if err = ESUCCESS then
          index_journal_update(key_1, adr_1);
          index_journal_update(key_2, adr_2);
          index_journal_update(key_3, adr_3);
```

Figure 11.5: Example Operation of Component *FFSC*: `ffsc_create` (precondition omitted)

Note that injectiveness is technically not required for the verification of the *Index&Journal* component, but it is necessary for the verification of its implementation as discussed in more detail in Sec. 11.5.

The subcomponent provides access to specific keys in the index (lookup, update and remove), an operation to remove all data keys above a certain page and an operation that queries all directory entries of a given directory. The latter two operations are needed to truncate a file and to get a listing of a directory.

The second set of operations provided by the subcomponent gives access to the journal and log. It is possible to read a node at a certain address, to perform a transaction that adds n-nodes for $1 \leq n \leq 5$ to the node store and the log, and to synchronize the contents of the node store and log with flash memory. A commit copies the current RAM index to the flash index, the RAM orphans to the flash orphans, empties the log and synchronizes the state with the flash device. The RAM orphans are also an instance of the *commit data structures* introduced in Ch. 6. The commit also removes some obsolete nodes from the node store as an additional form of garbage collection necessary for the index as discussed in Sec. 11.7.[1]

The recovery operation of *Index&Journal* restores the flash index and the flash orphans into the RAM state and returns the RAM/flash orphans as well as the current log. Based on this information the component *FFSC* replays the log and restores the previous RAM index where all orphaned files have been removed.

The invariant of component *Index&Journal* are trivially maintained given the preconditions of each of the operations.

Normal Operations of Component *FFSC* The operations of the component *FFSC* first perform some lookup of addresses and nodes in the RAM index. Then they prepare new (versions of) the nodes and write them to flash in one transaction yielding new addresses. Afterwards, the RAM index is updated to refer to the new versions. As example Fig. 11.5 shows the `ffsc_create` operation, which creates a new file given the parent inode number *parentino* and the new name of the file as a dentry object. First, some keys are prepared and

[1] Additionally, it is possible to query whether a commit is currently necessary, because the file system ran out of space to store the log. The storage mechanism for the log is discussed in more detail in Ch. 12. If this is the case the component *FFSC* can trigger a commit.

interface operations
`index_journal_gc()` **pre** `gc−cons`$(rindex, ns)$
// Move nodes to new addresses
choose $keylist$: `List`⟨`NodeKey`⟩, $adrlist$: `List`⟨`NodeAddress`⟩
with $\#\, keylist = \#\, adrlist \wedge \neg\, \texttt{dups}(keylist) \wedge \neg\, \texttt{dups}(adrlist)$
$\wedge\, \texttt{fresh}(adrlist, ns) \wedge keylist \subseteq dom(rindex)$
in
$ns := ns[adrlist, ns[rindex[keylist]]]$;
$log :+= adrlist$;
$rindex[keylist] := adrlist$;
// Remove obsolete nodes
choose $adrset$: `Set`⟨`NodeAddress`⟩ **with** $adrset \cap (ran(rindex) \cup log) = \emptyset$ **in**
$ns := ns \setminus adrset$;
$synced := false$;

Figure 11.6: Specification of Garbage Collection in the Component *Index & Journal* (Continuation of Fig. 11.4 on page 128)

the node that corresponds to the parent directory is loaded into $node_1$ and it's size, which reflects the number of entries in the directory, is increased. The other two nodes represent the directory entry and an inode node for the new file. Finally, a transaction with three nodes is performed and the index is adjusted accordingly.

All other interface operations are similarly implemented, except for commit, which just calls the commit operation of the *Index & Journal* component, and recovery.

Recovery & Replay of the Log (*FFSC*) The recovery from a power failure of component *FFSC* first calls the recovery of component *Index & Journal*, which returns the log and the flash orphans. The flash orphans are removed from the initial RAM index. Then the individual nodes referred to by the log are replayed one after the other in sequence. It is proven that this recovery routine results in the same state as an algebraic version `replay-log`$(log, ns, findex, forphans)$ of it. The additional Invariant (replay-log) of the component *FFSC* implies the correctness of its recovery operation.

invariant `replay-log`$(log, ns, findex, forphans) = rindex \ominus rorphans$ (replay-log)

The operation $\ominus$ removes all inode and data keys from the RAM index that belong to an orphaned file. The recovery then essentially corresponds to the crash behavior of the component *POSIX* of Ch. 7.

More details about the recovery proof and the proof of refinement of component *AFS* are given in Ernst [43, Ch. 9].

Specification of Garbage Collection (*Index & Journal*) Garbage collection is an internal operation of the component *FFSC* and triggered similarly to wear-leveling of Ch. 10, i.e., by a flag that is set by the other operations. The actual implementation, however, is part of the (implementation of the) *Index & Journal* component as a separate interface operation. The specification of garbage collection is depicted in Fig. 11.6. As a precondition, the current RAM index must be consistent with the node store, i.e., the Invariant (gc-cons) of *FFSC* must be satisfied. This ensures that the inverse mapping stored in nodes is consistent with the RAM index. Garbage collection uses this inverse mapping to distinguish live from obsolete objects and updates the correspond entries in the RAM index afterwards. Consistency of this inverse mapping with the RAM index guarantees that the correct mappings from keys to addresses are updated.

In the first step, the algorithm choses the nodes that need to be moved by their keys (variable *keylist*) and fresh addresses as a target for the copying (variable *adrlist*). Then the

nodes that correspond to the keys are moved to the new location, their addresses added to the log and the RAM index is updated accordingly. The lookup and update function on maps, _[_] and _[_ , _] respectively, are extended to lists of keys, addresses and nodes to facilitate the specification of garbage collection as shown in the figure.

In the second step, a set of allocated addresses, that are not referenced by the log or RAM index, are deallocated and removed from the node store. The implementation might buffer some of the operations, therefore the synchronized flag is reset at the end.

Note that the specification of garbage collection exhibits some details of the hardware and the implementation. In the context of just the *FFSC* and *Index&Journal* a valid approach to garbage collection would just remove unreferenced nodes from the node store. However, in a real implementation several nodes are stored within the same erase block and allowing garbage collection of the erase block only after all nodes in the block are obsolete would certainly impair proper utilization of free space. The implementation of garbage collection therefore first has to move referenced nodes to a new location and only then the erase block can be deallocated or unmapped.

The invariants of the component *Index&Journal* hold over each of the two steps individually. The only trouble during the verification is the interplay between the extended lookup and update functions, which do not admit easy lemmas, i.e., it is not clear how $ns[adrlist, ndlist][adr]$ can be simplified without referring to complicated functions that return the index of an address in a list and lookup at an index in a list. Only in the context that nodes are copied and an index is updated desirable lemmas are possible, such as

$$ns[adrlist, ns[rindex[keylist]]][rindex[keylist, adrlist][key]] = ns[rindex[key]]$$

under suitable preconditions that include $\texttt{gc-cons}(rindex, ns)$. This lemma simplifying the lookup at an address from the modified index in the altered node store is necessary in order to prove the Invariant (gc-cons) in the state after garbage collection.

The *Index&Journal* component is implemented in two steps as shown in Fig. 11.2 on page 124. In the first step transactions and the block structure of the flash hardware are introduced and the index remains an abstract, partial function. In component *Transactions* garbage collection is performed and the log is merged with the node store. In the second step the index is implemented as a wandering B^+-tree on top of the persistence layer. Sections 11.4 to 11.6 cover the implementation and verification of the *Transactions* and *Index&Persistence* components. The integration of the *B^+-tree* component into commit and recovery, and garbage collection of the index is briefly discussed in Sec. 11.7.

11.3 Persistence of Transactional Nodes

Transactional Node Store The *Transactions* component encapsulates the nodes of the journal in a transactional node `TNode`. A transactional node additionally contains two markers `begin` and `end`, which signify whether this node is the beginning and end of a transaction, respectively.[2]

data type $\texttt{TNode} = \texttt{tnode}(\texttt{node}\colon \texttt{Node}, \texttt{begin}\colon \mathbb{B}, \texttt{end}\colon \mathbb{B})$

Nodes belonging to a transaction are written sequentially within an erase block, therefore those two markers are sufficient to identify all nodes of a transaction and check whether a transaction is complete by looking for the node with an `end` marker set. Partially written transactions occur either due to hardware errors or a power failure.

[2] As an extension a transaction node could also store a sequence number in order to allow for a multi-headed log. For example UBIFS uses a log head for normal operations and another one for garbage collection in order to improve concurrency. Furthermore, error-correction codes could be added to transaction nodes in order to improve robustness and reliability in the event of hardware errors.

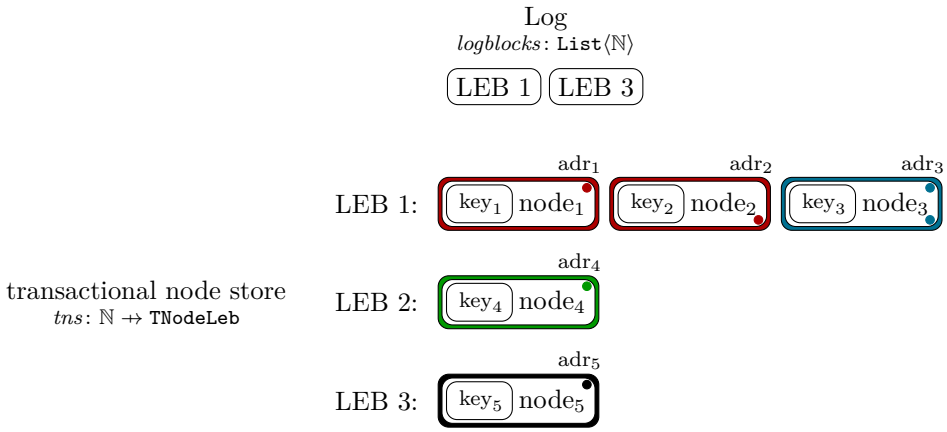

Figure 11.7: The transactional journal component introduces the block structure to the node store and the log and augments nodes with the details of the transaction they belong to, depicted by the thick, colored borders. The start marker and end marker are indicated by colored circles in the top-right and bottom-right corner, respectively. The basic operation on the erase blocks is appending transactional nodes at the current head of the log, i.e., the last block of the list *logblocks*.

The component *Index&Persistence* keeps these transactions nodes arranged in a block structure in the *transactional node store* (= *tns*), which maps a block number to the sequence of transactional nodes it contains.

state $tns: \mathbb{N} \nrightarrow$ `TNodeLeb`

Each individual erase block stores a list of transactional nodes `tnodes` and for each node its offset in bytes in the block in the field `toffs`.

data type `TNodeLeb` = `tnode-leb`(`tnodes`: `List`⟨`TNode`⟩,
`toffs`: `List`⟨ℕ⟩,
`ref-size`: ℕ)

We tacitly assume that both lists have the same length and that the list of offsets is always duplicate-free, in the actual model this is of course an additional invariant of the *Index&Persistence* component.[3] The field `ref-size` stores the number of bytes in the block that are still referenced in the RAM index. It is used to find suitable blocks for garbage collection efficiently and is explained in Sec. 11.5 further.

Fig. 11.7 depicts the transactional node store with 4 colored transactions and beginning and end markers. The **green** and **black** transactions are only partial, since they miss a corresponding node marking the end of the transaction in the same block.

The block structure of the *tns* necessitates that the node addresses of the previous section are refined.

Node Addresses An address of a (transactional) node is defined as a tuple consisting of the LEB number, the offset within the LEB and the size in bytes of the nodes. The size is stored in the address for two reasons. First, it is used to correctly update the `ref-size` field

[3]Note that at this level of abstraction it is not possible to calculate the offsets of the nodes, even if the size of a *node* is known as `serialized-size`⟨`T`⟩(*node*). The reason is that in between the byte-representations of two transactional nodes in the block there might be padding nodes to ensure that a page-aligned write is possible. This is discussed in more detail in Ch. 12.

of each LEB on each update of the index as detailed in Sec. 11.5. Second, it can be used to read the node with one I/O operation, instead of having to read a tiny header containing the size first and reading the complete node in a second I/O operations. Using additional I/O operations here would incur a lot of overhead and be quite expensive in terms of performance.

data type $\texttt{NodeAddress} = \texttt{address}(\texttt{leb}: \mathbb{N}, \texttt{off}: \mathbb{N}, \texttt{size}: \mathbb{N})$

The sequence of addresses of a LEB *tleb* with number l are given by the function

$$\texttt{addresses}: \mathbb{N} \times \texttt{TNodeLeb} \to \texttt{List}\langle\texttt{NodeAddress}\rangle$$

defined by Equation (leb-addresses). The helper function `addresses-h` is defined structurally recursive over the list of transactional nodes and their offsets.

$$\texttt{addrs}(l, \mathit{tleb}) = \texttt{addrs-h}(l, \mathit{tleb}.\texttt{tnodes}, \mathit{tleb}.\texttt{toffs}) \tag{leb-addresses}$$

$$\texttt{addrs-h}(l, [], []) = []$$
$$\texttt{addrs-h}(l, \mathit{tnode} + \mathit{tnodes}, \mathit{offset} + \mathit{offsets})$$
$$= \texttt{address}(l, \mathit{offset}, \texttt{serialized-size}\langle\texttt{T}\rangle(\mathit{tnode})) + \texttt{addrs-h}(l, \mathit{tnodes}, \mathit{offsets})$$

We define two shorthands that allow us to use the transactional node store *tns* similarly to the node store *ns* of the previous Sec. 11.2. An address *adr* is allocated in the transactional node store *tns*, written $\mathit{adr} \in \mathit{tns}$ and defined by Eqn (adr-in), if the LEB is allocated, the offset is known in the LEB and the size of the address matches the size of the encoded node, defined by the `serialized-size` function of Ch. 9. If an address *adr* is allocated in *tns*, the lookup $\mathit{tns}[\mathit{adr}]$ returns the corresponding transactional node. Lookup is defined by Eqn (at-adr).

$$\mathit{adr} \in \mathit{tns} \leftrightarrow \mathit{adr}.\texttt{leb} \in \mathit{tns} \wedge \mathit{adr} \in \texttt{addrs}(\mathit{adr}.\texttt{leb}, \mathit{tns}[\mathit{adr}.\texttt{leb}]) \tag{adr-in}$$
$$\mathit{tns}[\mathit{adr}] = \mathit{tns}[\mathit{adr}.\texttt{leb}].\texttt{tnodes}[\texttt{index-of}(\mathit{adr}.\texttt{off}, \mathit{tns}[\mathit{adr}.\texttt{leb}].\texttt{toffs})] \tag{at-adr}$$

Abstraction Relation Although we have only seen (part of) the state of component *Index&Persistence* and almost nothing of the implementation *Transactions*, it is already possible to understand the crucial part of the abstraction relation between the specification component *Index&Journal* and the other two components. The basic idea is that the node store can be determined by a function $\texttt{nodes}(\mathit{tns})$ that extracts the addresses and nodes from the transactional node store *tns* as defined by Equation (nodes).

$$\texttt{nodes}(\mathit{tns}) = \{\, \mathit{adr} \mapsto \mathit{tns}[\mathit{adr}].\texttt{node} \mid \mathit{adr} \in \mathit{tns} \,\} \tag{nodes}$$

However, there are two additional issues to tackle.

The first issue is that the result of $\texttt{nodes}(\mathit{tns})$ still contains partial transactions. Note that such transactions are always *at the end* of a block, because we write the block sequentially and have to discontinue writing once we have encountered a hardware failure. Note that in such cases the erase block manager of Ch. 10 already tried to move the (old and new) data to a new *physical* block and failed. Therefore, write failures are treated in components above the erase block manager as a fatal error, and a read-only mode is entered.

The function $\mathit{tns}\!\downarrow$ discards the partial transactions at the end of each block in *tns* and is defined by Equations (discard-partial-transactions). The notation $x[..n]$ denotes the prefix of the list x consisting of the first n elements of the list.

$$\mathit{tns}\!\downarrow = \{\, l \mapsto \mathit{tns}[l]\!\downarrow \mid l \in \mathit{dom}(\mathit{tns}) \,\} \tag{discard-partial-transactions}$$
$$\mathit{tleb}\!\downarrow = \texttt{tnode-leb}(\mathit{tleb}.\texttt{tnodes}[..\texttt{valid-tnodes}(\mathit{tleb}.\texttt{tnodes})]$$
$$\mathit{tleb}.\texttt{toffs}[..\texttt{valid-tnodes}(\mathit{tleb}.\texttt{tnodes})]$$
$$\mathit{tleb}.\texttt{ref-size})$$
$$\texttt{valid-tnodes}(\mathit{tnodes} + \mathit{tnode}) = (\ \mathit{tnode}.\texttt{end} \supset \#\,\mathit{tnodes} + 1\ ;\ \texttt{valid-tnodes}(\mathit{tnodes})\ )$$

In Fig. 11.7 on page 132 this removes the **green** and **black** nodes that belong to incomplete transactions.

The second issue is that deallocated LEBs might reappear after a power failure as discussed in Ch. 10 and picked up again in the next Ch. 12. The reason is that the set of LEBs allocated for the journal $dom(tns)$ and their `ref-size` field are stored in a *commit data structure* (see Ch. 6), called the *LEB Properties Array* (LPA) in Ch. 12. The set of deallocated LEBs is modeled by the state variable *invallebs* of the component *Index&Persistence*.

state $invallebs\colon \texttt{Set}\langle\mathbb{N}\rangle$

With these prerequisites the abstraction relation between the node store and the transactional node store is given by Equation (abs-node-store). First the invalid LEBs are removed, then all partial transactions are discarded and only afterwards the addresses and nodes are extracted.

abstraction relation $ns = \texttt{nodes}(\,(tns \setminus invallebs)\!\downarrow\,)$ (abs-node-store)

The node store *ns* corresponding to the situation depicted in Fig. 11.7 on page 132 according to the abstraction relation (abs-node-store) is $\{key_1 \mapsto node_1, key_2 \mapsto node_2, key_3 \mapsto node_3\}$.

The log of Sec. 11.2 is also represented differently in component *Index&Persistence* and is partly merged with the transactional node store. Only the sequence of LEB numbers *logblocks*, which form the log are stored explicitly.

state $logblocks\colon \texttt{List}\langle\mathbb{N}\rangle$

The log of the specification is recovered by the function $\texttt{log}(logblocks, tns)$ defined by Equation (log).

$$\texttt{log}([], tns) = [] \qquad \text{(log)}$$
$$\texttt{log}(l + logblocks, tns) = \texttt{addrs}(l, tns[l]) + \texttt{log}(logblocks, tns)$$

In the abstraction relation partial transactions also need to be discarded as shown by Equation (abs-log).

abstraction relation $log = \texttt{log}(logblocks, tns\!\downarrow)$ (abs-log)

Fig. 11.7 on page 132 depicts a situation where *logblocks* is the list [1, 3]. The abstraction (abs-log) yields the list $[adr_1,\ adr_2,\ adr_3]$ for *log*.

It is imperative to know that the LEBs forming the log are not invalid and Invariant (loglebs-valid) holds. Furthermore, the list should be duplicate-free.

invariant $logblocks \cap invallebs = \emptyset \wedge \neg\, \texttt{dups}(logblocks)$ (loglebs-valid)

Note that the abstraction to the log requires that the addresses are ordered. For this reason the data type `TNodeLeb` for blocks uses two lists for nodes and addresses at this level of abstraction, instead of a partial function of the type $\mathbb{N} \nrightarrow$ `TNode` mapping offsets to nodes. The latter approach would simplify lookup as defined by Equation (at-adr), but complicate the reconstruction of the log in Equation (log) and the removal of partial transactions in Equation (discard-partial-transactions).

Fig. 11.8 provides an overview over the complete state, the invariants and the interface operations concerned with journaling in the *Index&Persistence* component. The state variable *ftns* is part of the commit and recovery mechanism, explained in more detail in Sec. 11.6.

The component provides operations to query information about the log, i.e., determine whether it is empty and how many bytes are still available in the last block in the log. All transactions are appended to the last block in *logblocks* if there is enough space available. This block is usually called the current *head* of the journal. If necessary the nodes in the journal can be synchronized to disk. Finally, if no space is available the current journal head can be moved to a newly allocated erase block.

Sections 11.4 to 11.6 explain the components *Transactions* and *Index&Persistence* and prove that it refines the specification *Index&Journal*, summarized by Thm. 9.

component *Index&Persistence*
state $tns\colon \mathbb{N} \nrightarrow$ `TNodeLeb`, $invallebs\colon$ `Set`$\langle\mathbb{N}\rangle$, $logblocks\colon$ `List`$\langle\mathbb{N}\rangle$, $synced\colon \mathbb{B}$
$ftns\colon \mathbb{N} \nrightarrow$ `TNodeLeb`, $rindex, findex\colon$ `NodeKey` $\nrightarrow$ `NodeAddress`, $forphans\colon$ `Set`$\langle\mathbb{N}\rangle$
invariant
$logblocks \cap invallebs = \emptyset \wedge \neg$ `dups`$(logblocks) \wedge$ `inj`$(rindex) \wedge$ `inj`$(findex)$
$\wedge\ dom(tns) \cup invallebs = dom(ftns) \cup logblocks$ (Sec. 11.6)
interface operations
`index_persistence_read_tnode`$(adr; tnode, err)$ **pre** $adr \in tns \setminus invallebs$
$tnode := tns[adr]$
`index_persistence_is_journal_empty`$(; IsEmpty)$
$IsEmpty := logblocks = []$
`index_persistence_get_journal_head_freesize`$(; bytes)$
$bytes := ?$
`index_persistence_journal_head_append`$(tnodes; adrlist, err)$ **pre** $logblocks \neq []$
choose $toffs$ **with** $\#\ toffs \leq \#\ tnodes \wedge \neg$ `dups`$(tnodes[logblocks.$`last`$].$`toffs`$ + toffs)$ **in**
$tns[logblocks.$`last`$].$`tnodes` $:+= tnodes[..\#\ toffs]$;
$tns[logblocks.$`last`$].$`toffs` $:+= toffs$;
$adrlist :=$ `addrs-h`$(logblocks.$`last`$, tnodes[..\#\ toffs], toffs)$;
if $\#\ toffs = \#\ tnodes$ **then** $err :=$ `ESUCCESS` **else** `fail`$(; err)$
`index_persistence_sync`$(; err)$
$\{\ synced := true; err :=$ `ESUCCESS` $\}\ \vee\ \{$ `fail`$(; err)\ \}$
`index_persistence_move_log_head`$(; err)$ **pre** $synced$
choose l **with** $l \notin dom(tns)$ **in**
$tns[l] :=$ `tnode`$([], [], 0)$, $logblocks :+= l$, $invallebs :-= l$

Figure 11.8: Component *Index&Persistence*: State, Invariants & Journaling Operations (error-handling omitted)

Theorem 9 (Correctness & Crash-Safety of Journaling & Garbage Collection)**.** *The component Transactions refines the component Index&Journal using the forward simulation*

$$ns = \texttt{nodes}(\ (tns \setminus invallebs){\downarrow}\ ) \wedge log = \texttt{log}(logblocks, tns{\downarrow})$$

with the addition that the identically named state variables rindex, findex and forphans of component Index&Journal and component Index&Persistence are equal.

11.4 Journaling & Transactions

The component *Transactions* uses the interface provided by component *Index&Persistence* described in Sec. 11.3 to build transactions of `TNodes` and to append them to the journal head. Fig. 11.9 shows the fundamental, auxiliary operation `transactions_transaction`. It takes a list of nodes, each of which contains a file system object or a modification to it, and *atomically* appends them to the journal. Note that the component *Index&Persistence* does not provide the required atomicity. In the event of an error a prefix of the nodes might be persisted by the append operation in Fig. 11.8.

In the following the verification of the operation `transactions_transaction` is presented alongside the explanation of the operation itself.

First, the operation builds a list of transactional nodes via `transactions_compose` with the correct transaction begin and transaction end markers set. The result is a list *tnodes* that forms a *complete transaction for nodes*, written `complete`$(tnodes, nodes)$ and defined

component *Transactions*
subcomponent *Index&Persistence* (Fig. 11.8 on page 135)
state *romode*: $\mathbb{B}$,
invariant $\big(\neg\, romode \rightarrow \texttt{valid-journal-head}(logblocks, tns)\big)$
$\wedge\ \texttt{lebs}(rindex) \subseteq dom(tns) \wedge \texttt{lebs}(findex) \subseteq dom(ftns)$ (Sec. 11.5 and 11.6)
$\wedge\ \texttt{ref-size-inv}(rindex, tns) \wedge \texttt{ref-size-inv}(findex, ftns)$ (Sec. 11.5 and 11.6)
auxiliary operations
`transactions_transaction`(*nodes*; *adrlist*, *err*)
if *romode* **then**
err := `EROFS`;
else let *tnodes*, *size* **in**
`transactions_compose`(*nodes*; *tnodes*, *size*);
`transactions_allocate`(*size*; *err*);
if *err* = `ESUCCESS` **then**;
`index_persistence_journal_head_append`(*tnodes*; *adrlist*);
if *err* ≠ `ESUCCESS` **then**
romode := *true* // Enter read-only mode
interface operations
`transactions_get`(*adr*; *node*, *err*)
let *tnode* **in**
`index_persistence_read_tnode`(*adr*; *tnode*, *err*);
if *err* = `ESUCCESS` **then** *node* := *tnode*.`node`

Figure 11.9: Component *Transactions*: State, Invariants & Basic Operations for Journaling

by Equation (complete-transaction).

$$
\begin{aligned}
& \texttt{complete}(tnodes, nodes) && \text{(complete-transaction)}\\
\leftrightarrow\ & \#\, tnodes = \#\, nodes \wedge nodes \neq [] \wedge tnodes.\texttt{head.begin} \wedge tnodes.\texttt{last.end}\\
& \wedge \big(\ \forall i.\, 0 < i < \#\, tnodes \ \rightarrow \neg\, tnodes[i].\texttt{begin}\ \big)\\
& \wedge \big(\ \forall i.\, 0 \leq i < \#\, tnodes - 1 \rightarrow \neg\, tnodes[i].\texttt{end}\ \big)\\
& \wedge \big(\ \forall i.\, 0 \leq i < \#\, tnodes \ \rightarrow tnodes[i].\texttt{node} = nodes[i]\ \big)
\end{aligned}
$$

A list of transactional nodes *tnodes*′ is a *partial transaction*, written `partial`(*tnodes*′) and defined by Equation (partial-transaction), if the transaction begin marker is set appropriately, but no transaction end marker is set.

$$
\begin{aligned}
& \texttt{partial}(tnodes') && \text{(partial-transaction)}\\
\leftrightarrow\ & \big(tnodes' \neq [] \rightarrow tnodes.\texttt{head.begin}\big)\\
& \wedge \big(\ \forall i.\, 0 < i < \#\, tnodes \ \rightarrow \neg\, tnodes[i].\texttt{begin}\ \big)\\
& \wedge \big(\ \forall i.\, 0 \leq i < \#\, tnodes \rightarrow \neg\, tnodes[i].\texttt{end}\ \big)
\end{aligned}
$$

Any strict prefix *tnodes*′ of a complete transaction *tnodes* is a partial transaction. If an error occurs while persisting a complete transaction *tnodes*, then only a strict prefix of *tnodes* and therefore a partial transaction is actually written.

Afterwards, `transactions_allocate` (not shown) synchronizes the journal head and moves it to a new block if the journal is either empty or the journal head does not provide enough space for the transaction. Note that moving the journal head appends new elements to *logblocks* and allocates a new block in *tns*. However, the abstract view given by the abstraction relation (abs-node-store) and (abs-log) of Sec. 11.3 are left unchanged and Equations (allocate-pre-post) hold, where the unprimed and primed variables denote the

state before and after the operation `transactions_allocate`, respectively.

$$\texttt{nodes}(\,(tns' \setminus invallebs')\!\downarrow\,) = \texttt{nodes}(\,(tns \setminus invallebs)\!\downarrow\,) \quad \text{and} \quad \texttt{log}(logblocks', tns'\!\downarrow) = \texttt{log}(logblocks, tns\!\downarrow) \tag{allocate-pre-post}$$

In the next step an attempt is made to append the transactional nodes to the journal head. If this fails, the file system switches to read-only mode, by setting the corresponding state variable *romode*. If the file system is in read-only mode, then it will usually return the error code `EROFS` for write operations.

state $romode : \mathbb{B}$

invariant $\neg\, romode \rightarrow \texttt{valid-journal-head}(logblocks, tns)$

The file system can exit the read-only mode only by remounting after a reboot or power failure. As long as the file system is not in read-only mode, the journal head is *valid*, written `valid-journal-head`$(logblocks, tns)$ and defined by Equation (valid-journal-head).

$$\begin{aligned} &\texttt{valid-journal-head}(logblocks, tns) \qquad\qquad (\text{valid-journal-head}) \\ \leftrightarrow\ &\big(\ logblocks \neq [] \wedge tns[logblocks.\texttt{last}].\texttt{tnodes} \neq [] \\ &\quad \rightarrow tns[logblocks.\texttt{last}].\texttt{tnodes.last.end}\ \big) \end{aligned}$$

The last transactional node in a valid journal head, assuming there are any nodes at all, is always marked as an end transaction node. Otherwise, the last transaction could only be written partially and we should not continue the normal operations afterwards. Note that the erase block manager of Ch. 10 will already have tried to move the *logical* erase block of the journal head to a new *physical* erase block by atomically moving its contents, i.e., a lot of attempts to recover from this situation were already made and it is better to give up in this case. In practice, the user should run a file system check tool.

The verification basically has to prove that the writes propagate fully to the abstract view given by the abstraction relations (abs-node-store) and (abs-log) of Sec. 11.3 in case the operation returns successfully. In the case of a hardware failure the abstract view must be unchanged.

If unprimed variables denote the state before the execution of `transactions_transaction` and primed variables denote the state after allocation, the post-state of a successful run is basically given by

$$\texttt{nodes}((tns'[logblocks'.\texttt{last}, \texttt{tnode-leb}(\ldots + tnodes, \ldots + toffs, \ldots)] \setminus invallebs')\!\downarrow)$$

for a list of transactional nodes *tnodes*, which is a complete transaction for the input nodes. The dots "…" stand for the previous contents of the respective field of $tns'[logblocks'.\texttt{last}]$. The following equation then proves that the write is visible on the abstract level.

$$\begin{aligned} &\texttt{nodes}((tns'[logblocks'.\texttt{last}, \texttt{tnode-leb}(\ldots + tnodes, \ldots + toffs, \ldots)] \setminus invallebs')\!\downarrow) = \\ \overset{(1)}{=}\ &\texttt{nodes}((tns' \setminus invallebs')[logblocks'.\texttt{last}, \texttt{tnode-leb}(\ldots + tnodes, \ldots + toffs, \ldots)]\!\downarrow) = \\ \overset{(2)}{=}\ &\texttt{nodes}((tns' \setminus invallebs')\!\downarrow[logblocks'.\texttt{last}, \texttt{tnode-leb}(\ldots + tnodes, \ldots + toffs, \ldots)]) = \\ \overset{(3)}{=}\ &\texttt{nodes}((tns' \setminus invallebs')\!\downarrow)[\texttt{addrs-h}(logblocks'.\texttt{last}, tnodes, toffs), nodes] = \\ \overset{(4)}{=}\ &\texttt{nodes}((tns \setminus invallebs)\!\downarrow)[adrlist, nodes] \end{aligned}$$

Transformation (1) holds because $logblocks'.\texttt{last} \notin invallebs'$ according to the subcomponent Invariant (loglebs-valid) on page 134, which holds right after the allocation. In step (2) the knowledge that a complete transaction *tnodes* is appended is used to show that the write propagates over the removal of partial transactions performed by the operation $\downarrow$.

Note that this step also requires that the journal head is valid, i.e., ended in a complete transaction. Otherwise additional, previously filtered nodes would suddenly become visible. Transformation (3) propagates appending transactional nodes over the operation `nodes`, which extracts the addresses from *toffs* and the contents from the transactional nodes *tnodes*. The last step (4) is correct, because moving the journal head does not change the abstract view according to Equation (allocate-pre-post) on page 137 and the addresses that are calculated are the same as the addresses *adrlist* returned by the operation.

Similarly, it can be argued that if the operation is successful then

$$\begin{aligned} & \texttt{log}(logblocks', tns'[logblocks'.\texttt{last}, \texttt{tnode-leb}(\ldots + tnodes, \ldots + toffs, \ldots)\downarrow) = \\ = \ & \texttt{log}(logblocks, tns\downarrow) + adrlist \end{aligned}$$

also holds.

In the unsuccessful case, i.e., if only a strict prefix *tnodes*′ is appended to the journal head, then *tnodes*′ is only a partial transaction for the initial nodes. Instead of propagating over ↓ as in transformation (2) for a complete transaction, the nodes of a partial transaction are removed by transformation (2′).

$$\begin{aligned} & \texttt{nodes}((tns' \setminus invallebs')[logblocks'.\texttt{last}, \texttt{tnode-leb}(\ldots + tnodes', \ldots + toffs', \ldots)]\downarrow) = \\ \overset{(2')}{=} \ & \texttt{nodes}((tns' \setminus invallebs')\downarrow) = \cdots \end{aligned}$$

With these postconditions of the operation `transactions_transaction` the correctness of the journaling operations of component *Transactions* immediately follows.

Proof of Thm. 9 on page 135 (for Journaling Operations). The actual interface operations $\texttt{index\_journal\_transaction}_i$ for $i = 1 \ldots 5$ are implemented by the auxiliary operation `transactions_transaction` of the *Index&Journal* component of Sec. 11.2 (see Fig. 11.4 on page 128) by constructing lists of nodes as an input and deconstructing lists of addresses for the output. The refinement proofs for the interface operations directly follow from the above postconditions of `transactions_transaction`.

Reading a node from the journal is straightforward as shown in Fig. 11.9 on page 136. The address is used to read the transactional node *tnode* and afterwards the node itself is extracted from *tnode*, which is trivially correct by considering the abstraction relation. □

11.5 Garbage Collection

The out-of-place updates performed by the Flash File System Core lead to an accumulation of different versions of each file system object. All versions that are still in use are referenced by the RAM index *rindex*. All other versions are obsolete. Garbage collection attempts to remove obsolete versions in order to free and unmap logical erase blocks. Only then they can be re-used for the log.

From the perspective of functional correctness any LEB that is not currently part of the log can be garbage collected. However, the choice of the block is obviously crucial for the quality of the file system implementation. The natural choice is a block with the least number of referenced file system objects or, more accurately, a block with the least number of bytes that are still referenced by the RAM index.

Referenced Bytes in LEBs The *Index&Persistence* component provides an operation, which returns one of the blocks with the minimum `ref-size` field. This is also the block with the least amount of bytes referenced by the RAM index by Invariant (refsize-inv-rindex) of the *Transactions* component.

invariant $\texttt{ref-size-inv}(rindex, tns)$ (refsize-inv-rindex)

interface operations

`index_persistence_get_gc_block`$(; l, err)$
  choose l' **with** $l' \in dom(tns) \setminus logblocks$
    $\wedge\ tns[l'] = min\{\ tns[l''].\texttt{ref-size} \mid l'' \in dom(tns) \setminus logblocks\ \}$
  in
    $l := l'$

`index_persistence_get_refsize`$(l; refsize)$ **pre** $l \in dom(tns)$
  $refsize := tns[l].\texttt{ref-size}$

`index_persistence_set_refsize`$(l, refsize)$ **pre** $l \in dom(tns)$
  $tns[l].\texttt{ref-size} := refsize$

`index_persistence_read_leb`$(l; adrlist, tnodes, err)$ **pre** $l \in dom(tns) \setminus invallebs$
  $tnodes := tns[l].\texttt{tnodes}$, $adrlist := \texttt{addrs}(l, tns[l])$

`index_persistence_deallocate_leb`$(l; err)$ **pre** $l \in dom(tns) \setminus logblocks \wedge synced$
  $tns :-= l$, $invallebs :+= l$

Figure 11.10: Component *Index&Persistence*: Garbage Collection Operations (error-handling omitted)

with

$$\texttt{ref-size-inv}(rindex, tns)$$
$$\leftrightarrow\ \forall l \in dom(tns).\ tns[l].\texttt{ref-size} = \sum \{\ adr.\texttt{size} \mid adr \in ran(rindex) \wedge adr.\texttt{leb} = l\ \}$$

The invariant states that the `ref-size` field of LEB l stores the sum over all `size` fields of all addresses referenced by the RAM index that belong in LEB l. If all these addresses are also allocated in the transactional node store, which is the case as long as the component is not in the recovery phase, the `size` field of an address adr with $adr \in tns$ corresponds to the size of the node's byte representation according to the Definition (adr-in) on page 133 of $adr \in tns$.

Index Operations In order to maintain Invariant (refsize-inv-rindex) it is necessary to adjust the `ref-size` field on any update to $ran(rindex)$. If an address adr is removed from the index, the `ref-size` field is decreased by $adr.\texttt{size}$, and if the address is added it is increased by the same number. Note that maintaining this invariant requires that the index is injective, otherwise removing or adding addresses might not immediately translate to an update of the `ref-size` field if a second key still or already maps to the same address. Fortunately, the RAM index as well as the flash index are injective on the level of the *Index&Journal* component (see Fig. 11.4 on page 128) as well as on the level of the *Index&Persistence* component (see Fig. 11.8 on page 135).

In order to establish the precondition $l \in dom(tns)$ for calls to the *Index&Persistence* component that query and update the `ref-size` field (see Fig. 11.10), Invariant (rindex-lebs-cons) is necessary. It states that all LEBs of all addresses in the RAM index are also allocated in the transactional node store.

invariant $\texttt{lebs}(rindex) \subseteq dom(tns)$ (rindex-lebs-cons)

where

$$\texttt{lebs}(rindex) = \{\ l \mid \texttt{address}(l, \_, \_) \in ran(rindex)\ \}$$

Garbage Collection Algorithm Overview After the LEB for garbage collection is chosen, all its nodes and addresses are read. For each address it is checked whether it is

```
interface operations
  transactions_gc()
    pre gc−cons(rindex, nodes(tns \ invallebs↓))
    let l, err, size, adrlist, tnodes, nodes in
      index_persistence_get_gc_block(; l, err);
      if err = ESUCCESS then
        index_persistence_get_refsize(l; size);
        if size ≠ 0 then {
          index_persistence_read_leb(l; adrlist, tnodes, err);
          if err = ESUCCESS then transactions_referenced(l; adrlist, tnodes, nodes, err);
          if err = ESUCCESS then transactions_move(; nodes, err);
        }
        if err = ESUCCESS then index_persistence_sync(; err);
        if err = ESUCCESS then index_persistence_deallocate_leb(l; err);
auxiliary operations
  transactions_move(; nodes, err)
    while err = ESUCCESS ∧ nodes ≠ [] do
      let nodes_0, adrlist_0 in
        ... // Check the amount of space available at the journal head and select the
        ... // maximal prefix nodes_0 of nodes that fits in the available space
        transactions_transaction(nodes_0; adrlist_0, err);
        if err = ESUCCESS then transactions_update_index(; nodes_0, adrlist_0);
```

Figure 11.11: Component *Transactions*: Implementation of Garbage Collection (continuation of Fig. 11.9 on page 136)

still referenced by the index. All those nodes need to be moved to a new location, i.e., they are appended to the journal with a transaction as in Sec. 11.4. Afterwards, the index is updated and finally the old LEB is deallocated. All the necessary supporting operations of the *Index&Persistence* component are shown in Fig. 11.10. Note that the choice of LEB for garbage collection takes the quality criterion discussed above into account and must exclude all LEBs that are still in the log. The reason is that all nodes in the log need to be replayed during a recovery from power loss as discussed in Sec. 11.2. More formally, these LEBs are needed to maintain Invariant (replay-log) on page 130.

The implementation of the garbage collection algorithm is depicted in Fig. 11.11. In the following the correctness of the algorithm with respect to its specification as shown in Fig. 11.6 on page 130 is discussed alongside some of the implementation details. Thm. 10 is immediately clear from the above discussion.

Theorem 10 (Quality of Garbage Collection)**.** *The garbage collection algorithm of component Transactions chooses a logical erase block with the least amount of bytes still referenced by the RAM index.* □

Validity of the Target LEB After the choice of block, the algorithm first determines whether there are nodes to move by checking whether the `ref-size` field is zero. Note that this appears to be an optimization at first, however, actually this check establishes that the LEB l is not invalid, i.e., that $l \notin invallebs$ holds. As discussed in Sec. 11.6 *invallebs* contains all deallocated and unmapped LEBs that re-emerge after a power failure and should not be used. If the `ref-size` fields is non-zero, however, it is known that there is some address in the index which refers to the LEB. With the precondition that $\texttt{gc−cons}(rindex, \texttt{nodes}(tns \setminus invallebs\downarrow))$ (see (gc-cons) on page 127) holds, it is known that each LEB referenced by the RAM index is valid, since the abstraction relation removes invalid LEBs.

Only once it is established that the LEB is valid, reading its nodes and addresses is permitted as shown by the precondition of the corresponding operation in Fig. 11.10. Reading

the nodes yields the result

$$adrlist = \texttt{addrs}(l, tns[l].\texttt{toffs}) \qquad \text{and} \qquad tnodes = tns[l].\texttt{tnodes}$$

according to the specification of the read operation. Afterwards, all nodes *nodes* still referenced by the index are extracted by the operation `transactions_referenced` (not shown) by checking whether the mapping $tnodes[i].\texttt{key} \mapsto adrlist[i]$ is still currently part of the index for each i. Note that this step also requires the precondition $\texttt{gc-cons}(rindex, \texttt{nodes}(tns \setminus invallebs\downarrow))$, otherwise the node could be found under a key different from $tnodes[i].\texttt{key}$. The nodes returned by this check can be characterized as

$$nodes = ns[\texttt{ref-addrs}(\texttt{addrs}(l, tns[l].\texttt{toffs}), rindex)]$$

for $ns \equiv \texttt{nodes}(tns \setminus invallebs\downarrow)$ and where $\texttt{ref-addrs}(adrlist, rindex)$ is recursively defined and returns all addresses $adr \in adrlist$ with $adr \in ran(rindex)$ in the order of *adrlist*. Thus, the variable *nodes* contains all nodes that need to be copied.

Node Migration The auxiliary operation `transactions_move` copies the nodes to the journal as shown (conceptually) in Fig. 11.11. Iteratively the operation selects a prefix $nodes_0$ of the nodes that fits the space available at the current journal head and then writes them as a transaction as discussed in Sec. 11.4. As many nodes as possible are gathered in one transaction in order to only perform the minimum amount of I/O operations required. After each call to `transactions_transaction` the abstract view changes from

$$ns \quad \text{to} \quad ns[adrlist_0, nodes_0] \qquad \text{and from} \qquad log \quad \text{to} \quad log + adrlist_0.$$

again for $ns \equiv \texttt{nodes}(tns \setminus invallebs\downarrow)$ and $log \equiv \texttt{log}(logblocks, tns\downarrow)$. Afterwards, the operation `transactions_update_index` updates the index for each node, i.e., the current mapping of $nodes[i].\texttt{key}$ is replaced with $nodes_0[i].\texttt{key} \mapsto adrlist_0[i]$ for each i within bounds. The *rindex* is updated from

$$rindex \quad \text{to} \quad rindex[\texttt{keys}(nodes_0), adrlist_0]$$

where the function `keys` just lifts the selector `key` on nodes to lists of nodes.

The entire operation `transactions_move` has the post-condition (move-post).

$$\begin{aligned} \exists nodes_0, adrlist_0.\quad & nodes_0 \sqsubseteq nodes \wedge (err = \texttt{ESUCCESS} \leftrightarrow nodes_0 = nodes) \\ & \wedge \# nodes_0 = \# adrlist_0 \wedge \neg \texttt{dups}(adrlist_0) \wedge \texttt{fresh}(adrlist_0, ns) \\ & \wedge ns' = ns[adrlist_0, nodes_0] \wedge log' = log + adrlist_0 \\ & \wedge rindex' = rindex[\texttt{keys}(nodes_0), adrlist_0] \end{aligned} \qquad \text{(move-post)}$$

$$\begin{aligned} \text{where}\quad & nodes \equiv ns[\texttt{ref-addrs}(\texttt{addrs}(l, tns[l].\texttt{toffs}), rindex)] \\ & ns \equiv \texttt{nodes}(tns \setminus invallebs\downarrow) \\ & log \equiv \texttt{log}(logblocks, tns\downarrow) \end{aligned}$$

We now have the first part of the refinement proof for garbage collection.

Proof of Thm. 9 on page 135 (for Garbage Collection, Part 1). This post-condition corresponds to the first **choose** statement of the specification of garbage collection shown in Fig. 11.6 on page 130. A **choose** statement is an existential quantifier in a refinement proof and is instantiated as follows. For the variable *keylist* of keys of nodes that were moved and the resulting addresses *adrlist* the instance $\texttt{keys}(nodes_0)$ and $adrlist_0$ is chosen, respectively. This instantiation needs to satisfy the following two conditions:

1. $\texttt{keys}(\texttt{nodes}(tns \setminus invallebs\downarrow)[\texttt{ref-addrs}(\ldots, rindex)]) \subseteq dom(rindex)$

2. $\neg\,\texttt{dups}(\texttt{keys}(\texttt{nodes}(\mathit{tns} \setminus \mathit{invallebs}{\downarrow})[\texttt{ref-addrs}(\ldots, \mathit{rindex})]))$

Both conditions follow from the fact that the keys stored in referenced nodes correspond to the keys they are referred to by in the RAM index according to the Invariant (gc-cons) on page 127. Thus, these keys 1) are actually part of the index and 2) do not contain duplicates. The assignments to the variables *log* and *rindex* of the specification in Fig. 11.6 on page 130 then correspond to the equations of the post-condition (move-post). The assignment to *ns* in the specification yields

$$\begin{aligned} & \mathit{ns}\big[\mathit{adrlist}_0, \mathit{ns}\big[\underbrace{\mathit{rindex}[\texttt{keys}(\mathit{ns}[\texttt{ref-addrs}(\ldots, \mathit{rindex})])]}_{=\,\texttt{ref-addrs}(\ldots,\mathit{rindex})}\big]\big] = \\ = \;& \mathit{ns}[\mathit{adrlist}_0, \mathit{ns}[\texttt{ref-addrs}(\ldots, \mathit{rindex})]] \end{aligned}$$

with the given instantiation for *adrlist* and *keylist*. Note that the double lookup of keys of referenced nodes in the RAM index can be rewritten to just the calculation of the referenced addresses. Again, this only holds due to the Invariant (gc-cons). □

LEB Deallocation After all referenced nodes are migrated successfully to the current journal head the garbage collection algorithm of Fig. 11.9 deallocates and unmaps the logical erase block *l*.

Proof of Thm. 9 on page 135 (for Garbage Collection, Part 2). In the abstract view this removes all nodes in the LEB *l* that are part of a complete transaction, or more formally the nodes at addresses

$$\mathit{adrset} \equiv \left\{ \mathit{adr} \;\middle|\; \mathit{adr} \in \mathit{dom}(\texttt{nodes}(\mathit{tns} \setminus \mathit{invallebs}{\downarrow})) \wedge \mathit{adr}.\texttt{leb} = l \right\}$$

are removed from the domain of the node store *ns*. Therefore, this is the chosen instance for the existentially quantified variable *adrset* in the second **choose** statement of the specification in Fig. 11.6 on page 130, which satisfies the additional condition

$$\begin{aligned} & \mathit{adrset} \cap (\mathit{ran}(\mathit{rindex}') \cup \mathit{log}') \equiv \\ \equiv\;& \mathit{adrset} \cap \big(\mathit{ran}(\mathit{rindex}[\texttt{keys}(\mathit{nodes}_0), \mathit{adrlist}_0]) \cup \mathit{log} \cup \mathit{adrlist}_0\big) = \\ =\;& \mathit{adrset} \cap \Big(\big(\mathit{ran}(\mathit{rindex}) \setminus \mathit{rindex}[\texttt{keys}(\mathit{nodes}_0)]\big) \cup \mathit{adrlist}_0 \cup \mathit{log} \cup \mathit{adrlist}_0\Big) = \\ =\;& \mathit{adrset} \cap \Big(\big(\mathit{ran}(\mathit{rindex}) \setminus \mathit{adrset}\big) \cup \mathit{log} \cup \mathit{adrlist}_0\Big) = \mathit{adrset} \cap (\mathit{log} \cup \mathit{adrlist}_0) = \emptyset \end{aligned}$$

of the **choose** statement. The last intersection is empty, because 1) $l \notin \mathit{log}$ holds according to the choice of the target block and 2) the addresses $\mathit{adrlist}_0$ are fresh, distinguishing them from the addresses *adrset*. □

Taken together this shows that the implementation of the garbage collection algorithm refines its specification and is therefore correct.

11.6 Commit, Power Failures and Recovery

The commit persists the commit data structures—the index *rindex*, the orphans *rorphans* and the *LEB Property Array* (= LPA). The LPA is managed by the implementation of the component *Index & Persistence* and stores the allocation status of each LEB and its `ref-size` field. Storing the LPA only during a commit, however, has an effect on the component *Transactions*. More formally, during a power failure all LEBs that were deallocated since the last commit (and not yet reallocated) re-appear and for all LEBs the `ref-size` field is reset to its value at the point of the last commit. In order to express this effect the

interface operations
`index_persistence_commit`(*rorphans*; *err*)
pre *synced* $\wedge$ *dom*(*tns* $\setminus$ `unref-lebs`(*tns*)) $\cap$ *invallebs* $= \emptyset$
tns $:=$ *tns* $\setminus$ `unref-lebs`(*tns*), *logblocks* $:= []$, *invallebs* $:= \emptyset$;
findex $:=$ *rindex*, *forphans* $:=$ *rorphans*;
ftns $:=$ *tns*;
crash
$tns' =$ `revert-refsize`(*tns*, *ftns*, *invallebs*, *logblocks*)
$\wedge$ *invallebs*$'$ $=$ *invallebs* $\wedge$ *logblocks*$'$ $=$ *logblocks* $\wedge$ *ftns*$'$ $=$ *ftns* $\wedge$ *findex*$'$ $=$ *findex*
$\wedge$ *forphans*$'$ $=$ *forphans*
recovery
`index_persistence_recover`(; *logblocks*$'$, *forphans*$'$, *err*)
rindex $:=$ *findex*, *logblocks*$'$ $:=$ *logblocks*, *forphans*$'$ $:=$ *forphans*, *synced* $:=$ *true*;

Figure 11.12: Component *Index&Persistence*: Commit, Crash & Recovery (error-handling omitted)

component *Index&Persistence* memorizes the state of the transactional node store *tns* at the point of the last commit in the state variable *ftns*.

state *ftns*: $\mathbb{N} \pfun$ `TNodeLeb`

The relationship between the transactional node store *tns*, its version *ftns* of the last commit, the log and the invalid LEBs is characterized by Invariant (lebs-inv).

$$\textbf{invariant}\quad dom(tns) \cup \underbrace{invallebs}_{\substack{\text{deallocated LEBs}\\ \text{without reallocation}}} = dom(ftns) \cup \underbrace{logblocks}_{\text{allocated LEBs}} \qquad \text{(lebs-inv)}$$

The left-hand and right-hand side of Equation (lebs-inv) both capture the total set of LEBs allocated between the current state of the component and the last commit, just starting from different points in time. Note that none of these union operations is necessarily disjoint, because reallocation is possible, i.e., a block is first deallocated and afterwards allocated again. The only other constraint is

$$logblocks \cap invallebs = \emptyset$$

and is already given by Invariant (loglebs-valid) on page 134.

Power Failure Fig. 11.12 shows the specification of a power failure of *Index&Persistence*. The RAM index *rindex* is arbitrary. For the transactional node store *tns* the effect of a reversion of the commit data structures is captured by the predicate `revert-refsize` and defined by Eqn. (revert-journal-refsize).

$$\begin{aligned} &\texttt{revert-refsize}(tns, ftns, invallebs, logblocks) \qquad \text{(revert-journal-refsize)}\\ = \{\, l \mapsto \texttt{tnode-leb}(tnodes, toffs, refsize) \mid\ & l \in dom(tns) \cup invallebs\\ & \wedge\ refsize = (l \in dom(ftns) \rhd ftns[l].\texttt{ref-size}\,;0)\\ & \wedge\ \big(l \in dom(tns) \rightarrow\ tnodes = tns[l].\texttt{tnodes}\\ & \qquad\qquad \wedge\ toffs = tns[l].\texttt{toffs}\big)\ \} \end{aligned}$$

Note that the domain of *tns* after the power failure is the union of the domain prior to it and the invalid LEBs *invallebs*, i.e., corresponds to all LEBs that were at some point allocated between the last commit and the state prior to the crash.

Commit The commit of the component *Index&Persistence* persists the current state of the commit data structures as shown in Fig. 11.12. Additionally, it cleans up the remaining invalid LEBs. These LEBs have a value of 0 in their `ref-size` field.

$$\texttt{unref-lebs}(tns) = \left\{\, l \;\middle|\; l \in dom(tns) \wedge tns[l].\texttt{ref-size} = 0 \,\right\}$$

This is an overapproximation of all invalid LEBs. It is used, because the implementation of the *Index&Persistence* component has no access to the set of invalid LEBs *invallebs* since it is only a specification artifact.

The commit operation of the surrounding component *Transactions* synchronizes the journal head to ensure that all transactions are persisted, before it calls the commit operation of component *Index&Persistence*.

The commit transfers the Invariants (refsize-inv-rindex) on page 138 and (rindex-lebs-cons) on page 139 about the RAM data structures to their corresponding persistent state. This establishes the Invariants (refsize-inv-findex) on page 144 and (findex-lebs-cons) on page 144 of the component *Transactions*.

invariant

$$\begin{aligned} &\texttt{ref-size-inv}(findex, ftns) && \text{(refsize-inv-findex)}\\ \wedge\ &\texttt{lebs}(findex) \subseteq dom(ftns) && \text{(findex-lebs-cons)} \end{aligned}$$

After the recovery the Invariants (refsize-inv-findex) and (findex-lebs-cons) reestablish the Invariants (refsize-inv-rindex) and (rindex-lebs-cons).

Proof of Thm. 9 on page 135 (for Commit). The specification of the commit in the component *Index&Journal* (see Fig. 11.4 on page 128) allows for the deallocation of a set of addresses that is not referenced by the RAM index. For the removed addresses *adrset* the overapproximation

$$adrset \equiv dom\big(\texttt{nodes}\big(tns \setminus \underbrace{\big(dom(tns) \setminus \texttt{unref-lebs}(tns)\big)}_{\text{referenced LEBs}}\big)\big)$$

is chosen. The refinement proof is then straightforward and shows that the commit operation is implemented correctly. □

Recovery The recovery of the subcomponent *Index&Persistence* is shown in Fig. 11.12. It restores the flash index as the current RAM index and returns the flash orphans and all blocks that form the log $logblocks'$.

Fig. 11.13 depicts the recovery of the component *Transactions*, which restores the log of addresses log' from the log of LEB numbers $logblocks'$. The operation reads the addresses and transactional nodes of every LEB in the order of $logblocks'$ and afterwards discards addresses and nodes of partial transactions.

Proof of Thm. 9 on page 135 (for Crash & Recovery). The operation restores the version of the log prior to the power failure according to the abstraction relation for the log (abs-log) on page 134. The abstraction relation (abs-log) for the log and the node store (abs-node-store) on page 134 are unaffected by the crash transition, i.e., the equations

1. $\texttt{log}(logblocks, \texttt{revert-refsize}(tns, ftns, invallebs, logblocks)\!\downarrow) = \texttt{log}(logblocks, tns\!\downarrow)$
2. $\begin{aligned}&\texttt{nodes}(\,(\texttt{revert-refsize}(tns, ftns, invallebs, logblocks) \setminus invallebs)\!\downarrow\,)\\ =\ &\texttt{nodes}(\,(tns \setminus invallebs)\!\downarrow\,)\end{aligned}$

hold. The LEBs of *logblocks* are disjoint from the invalid LEBs *invallebs* according to Invariant (loglebs-valid) on page 134, establishing the first equation. The second equation follows from the insight that the abstraction removes $(\_ \setminus invallebs)$ all the LEBs that re-emerge with arbitrary contents, and all other LEBs retain their original contents. □

```
recovery
  transactions_recover(; log′, forphans′, err)
    romode := false, log′ := [];
    let logblocks′, adrlist, nodes in
      index_persistence_recover(; logblocks′, forphans′, err);
      if err = ESUCCESS then
        while err = ESUCCESS ∧ logblocks′ ≠ [] do
          index_persistence_read_leb(logblocks′.head; adrlist, nodes, err);
          if err = ESUCCESS then
            transactions_remove_partial_transactions(; adrlist, nodes);
            log′ :+= adrlist, logblocks′ := logblocks′.tail;
auxiliary operations
  transactions_remove_partial_transactions(; adrlist, nodes)
    let done = false in
      while ¬ done ∧ nodes ≠ [] do
        if nodes.last.end then
          done := true;
        else
          adrlist := adrlist.removelast, nodes := nodes.removelast;
```

Figure 11.13: Component *Transactions*: Recovery

As discussed in Sec. 11.2 after the recovery of the component *Transactions*, the log is replayed by component *FFSC*. This restores the RAM index prior to the power failure using the log and the flash index. This replay also restores the values of the field `ref-size`. Afterwards, all invalid LEBs have a `ref-size` field of 0 and are again removed, either by the garbage collector or at the latest by the next commit.

11.7 Index: A Wandering B$^+$-Tree

The index is implemented as a wandering B$^+$-tree [59, 33]. This section focuses on the integration of the index into the commit and recovery and its garbage collection, the verification of functional correctness is not in the scope of this thesis.

Each (in-memory) *node* of the B$^+$-tree has an associated *index node* on flash, which stores its contents. A non-leaf node of the tree stores an address to the index node of each child, in addition to the usual pointer to the child node. The address of the root node *froot* is stored in the superblock during the commit. The state variable *rroot* stores the pointer to the root node in main memory.

```
state  froot: NodeAddress,    rroot: Ref
```

Fig. 11.14 shows an example of a wandering B$^+$-tree with two pointers per node and the underlying flash device.

The B$^+$-tree is loaded lazily, i.e., after mounting the pointer to the root in main memory *rroot* is initialized to `null` and only *froot* node is loaded. During tree traversal if the pointer to a child node is `null` the corresponding index node is loaded into main memory first.

The persistent version of the tree is *wandering* in order to cope with the limitations of flash hardware that overwriting an index node is not possible. Index nodes are updated out-of-place and changed parts of the index move to new addresses at the point of the commit.

The component *B$^+$-tree* uses the subcomponent *Persistence* to store these index nodes. The subcomponent is similar to component *Index&Persistence* of Sec. 11.3, only the RAM and flash index are replaced by an *index node store*, given by the state variable *ins* and its

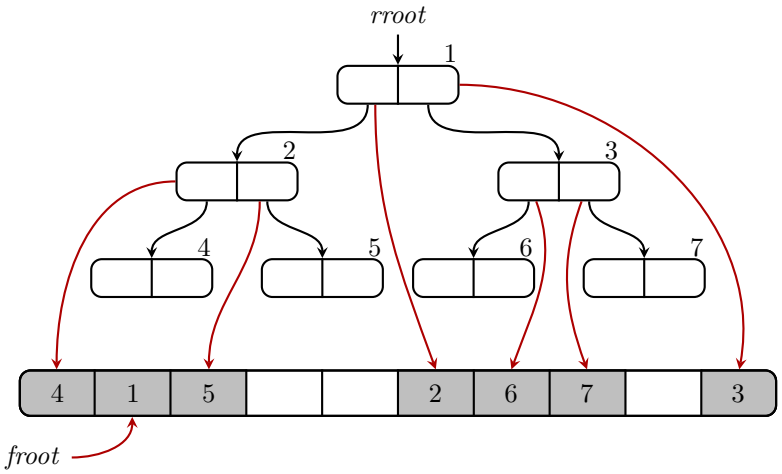

Figure 11.14: Wandering B$^+$-Tree: Black and red arrows denote pointers to main memory and flash addresses, respectively. The state directly after a commit is depicted, where all nodes in main memory correspond to their on-flash representation. Once a node in main memory is updated all flash pointers up to the root become invalid and during the next commit all nodes on the path from modified nodes to the root must be updated out-of-place. The numbers inside flash blocks correspond to the node in main memory they currently hold.

version of the last commit *fins*.

state $ins\colon \mathbb{N} \nrightarrow \texttt{INodeLeb}, \quad fins\colon \mathbb{N} \nrightarrow \texttt{INodeLeb}$

The LEB data structure `INodeLeb` itself is similar to the `TNodeLeb`s of the transactional journal, specifically there is also a field `ref-size` for each LEB, tracking the number of bytes still used. During a power failure all nodes allocated since the last commit in *ins* are removed, for all other nodes the field `ref-size` of *fins* is restored with the nodes of *ins*. The effect on all other state variables is the same as in component *Index & Persistence*, indicated by the dots.

crash $ins' = \texttt{revert-refsize}(ins, fins) \wedge \ldots$

with

$$\begin{aligned} &\texttt{revert-refsize}(ins, fins) && \text{(revert-index-refsize)}\\ = \; &\{\, l \mapsto \texttt{inode-leb}(inodes, ioffs, refsize) \mid && l \in dom(fins)\\ &&& \wedge\, refsize = ftns[l].\texttt{ref-size}\\ &&& \wedge\, inodes = ins[l].\texttt{inodes} \wedge ioffs = ins[l].\texttt{ioffs} \,\} \end{aligned}$$

Invariant (index-crash-inv) ensures that no critical index nodes are lost. It states that each LEB l allocated in *fins* is also allocated in *ins* and that the nodes and offsets of $fins[l]$ are a prefix of those in $ins[l]$.

invariant $fins \sqsubseteq ins$ (index-crash-inv)

The flash index is written during each commit. In the event of a successful commit the component *Persistence* removes all LEBs with an `ref-size` field of 0 in order to deallocate unneeded LEBs. The effect of a commit on *ins* and *fins* is therefore essentially the two

assignments

$$ins \;:=\; ins \setminus \texttt{unref-lebs}(ins);$$
$$fins \;:=\; ins;$$

Similar to Invariant (refsize-inv-rindex) for the RAM index, the `ref-size` field of LEB l must correspond to the sum of all sizes of all addresses that are still in use in the LEB l. The B$^+$-tree abstraction (details omitted)

$$\texttt{abs-btree}(rroot, froot, ins, \ldots, adrset)$$

calculates a set of index node addresses *adrset* that are currently referenced by the B$^+$-tree. With this set the Invariant (refsize-inv-btree) is expressible and ensures the correctness of the `ref-size` field.

invariant (refsize-inv-btree)

$$\forall l \in dom(ins).\ ins[l].\texttt{ref-size} = \sum \{\, adr.\texttt{size} \mid adr \in adrset \wedge adr.\texttt{leb} = l \,\}$$

Garbage collection of the flash index can be accomplished by marking all nodes as dirty, which ensures that the flash index is persisted completely in a new location and afterwards triggering a commit. The commit ensures that all LEBs of the previous flash index are deallocated, because their `ref-size` field is now 0.

Finally, the component *Persistence* has an explicit state for the current location of the write-buffer. Either the write-buffer is not in use, or writes to the journal head or to one of the LEBs for index nodes are buffered.

state $wbufstate$: `WBufState`

with

data type `WBufState` = `unbuffered` | `journalhead` | `index`(`leb`: $\mathbb{N}$)

Fig. 11.15 summarizes the specification component *Persistence* implemented in Ch. 12. The operations `persistence_move_log_head` and `persistence_allocate_index_leb` are used to allocate a new LEB for the journal or index and move the write-buffer to this LEB.

11.8 Related Work

Hesselink and Lali [63] specify a garbage collector in a model that maps file identifiers to nodes. Garbage collection removes file identifiers that are no longer reachable by a path. This is above the level of abstraction of the component *FFSC* and does not consider that nodes are stored at addresses and garbage collection might also need to move nodes.

Bilbyfs [11, 77, 9] deliberately has a slightly simpler design than UBIFS and Flashix. For example the index and several other data structures are currently not stored on flash and must be reconstructed during recovery by scanning the device. The focus of their research is the verification of the step from the models towards generated C code with their tool Cogent. The verification of the models in [9] is limited to a subset of the operations and does not consider crash-safety.

The two other verified file system FSCQ [27, 28, 26] and Yxv6 [123] are for magnetic disks only and do not feature garbage collection or an index, or have to consider the write characteristics of flash hardware, i.e., they can overwrite sectors of the disk. However, both file system employ a journal that records changes to sectors of the disk. FSCQ also groups individual changes to sectors into transactions, which are written to a sequential journal. Once the journal fills up, the changes are applied to the corresponding disk blocks, i.e., the sectors are overwritten. After a power failure, the recovery procedure scans the journal and

component *Persistence*
state $tns, ftns : \mathbb{N} \nrightarrow$ `TNodeLeb`, $invallebs$: `Set`$\langle\mathbb{N}\rangle$, $logblocks$: `List`$\langle\mathbb{N}\rangle$, $synced : \mathbb{B}$
$ins, fins : \mathbb{N} \nrightarrow$ `INodeLeb`, $froot$: `NodeAddress`, $forphans$: `Set`$\langle\mathbb{N}\rangle$
$wbufstate$: `WBufState`
invariant
$logblocks \cap invallebs = \emptyset \wedge \neg$ `dups`$(logblocks)$
$\wedge\ dom(tns) \cup invallebs = dom(ftns) \cup logblocks \wedge fins \sqsubseteq ins$
interface operations
// Operations for Journaling (see Fig. 11.8 on page 135)
`persistence_read_tnode`$(adr; tnode, err)$
`persistence_is_journal_empty`$(; IsEmpty)$
`persistence_get_journal_head_freesize`$(; bytes)$
`persistence_journal_head_append`$(tnodes; adrlist, err)$
`persistence_sync`$(; err)$
`persistence_move_log_head`$(; err)$
// Operations for Garbage Collection (see Fig. 11.10 on page 139)
`persistence_get_gc_block`$(; l, err)$
`persistence_read_leb`$(l; adrlist, tnodes, err)$
`persistence_deallocate_leb`$(l; err)$
// Index Operations
`persistence_allocate_index_leb`$(; err)$
`persistence_read_index_node`$(adr; indexnode, err)$
`persistence_index_leb_append`$(indexnode; adr, err)$
// Common Operations
`persistence_commit`$(froot', forphans'; err)$ (see Fig. 11.12 on page 143)
`persistence_get_refsize`$(l; refsize)$
`persistence_set_refsize`$(l, refsize)$
crash
$tns' =$ `revert-refsize`$(tns, ftns, invallebs, logblocks) \wedge invallebs' = invallebs$
$\wedge\ ftns' = ftns \wedge ins' =$ `revert-refsize`$(ins, fins) \wedge logblocks' = logblocks$
$\wedge\ froot' = froot \wedge forphans' = forphans$
recovery
`persistence_recover`$(; logblocks', forphans', froot', err)$

Figure 11.15: Component *Persistence*: State, Invariants & Interface

applies all pending writes to the actual blocks. The Yxv6 file system [123] uses a journaling technique similar to FSCQ.

Two other developments actually connect a high-level view to the pages and blocks of flash hardware [75, 36]. In both cases, only file content is mapped, written, and garbage collected at the granularity of flash pages, at the expense of extra state that is kept in memory. An encoding of the directory/file structure and the commit data structures down to flash and write-back caching are not considered. [75] deals with crashes during a write operation only and intertwines the recovery strategy with the implementation of the write operation. The models [75, 36] have a page-based allocation scheme assuming additional, overwritable bits that track the allocation status. These are not always present or might be used entirely for error-correction codes [131]. We have to recover newly allocated blocks and deallocate reappearing blocks after a power cut as shown by Sec. 11.6. This complicates the invariants, but makes the approach applicable to a wider set of flash hardware. The models [75, 36] do not consider the restriction to sequential writes within an erase block. [36] reads all pages during mounting/recovery in order to rebuild the index.

The use of synchronized states as a specification mechanism for order-preserving write-back caches and their use to ease the burden of specification in the context of a journaling file system

is novel. It simplifies the specification components *Index*&*Journal* and *Index*&*Persistence* and *Persistence* of this chapter significantly, as well as all components above them in the refinement hierarchy. Otherwise, in the event of a power failure it is necessary to specify which (transactional) nodes 1) are removed and 2) may be removed explicitly, as we initially did in our own previous work [46].

12

Write-Buffering, Node Persistence & Commit Atomicity

Summary. This chapter fills the gap between the transactional journal and index of Ch. 11 and the erase block manager of Ch. 10. The serialization of the transactional and index nodes must ensure that they appear to be written atomically, although the error model of Ch. 8 permits partial writes and power failures may occur at any time during the execution. Additionally, the specific write characteristics of flash hardware need to be taken into consideration: writes of the byte representation are cached until a page-aligned prefix of the data can be written. Writes must be sequential within each erase block, too. The write-buffer is the component where retractions and synchronized states of Ch. 4 are introduced into the refinement hierarchy with a crash refinement of Ch. 5. Additionally, critical data structures such as the LPA (= LEB Property Array), the sequence of log blocks and the flash orphans of Ch. 11 need to be updated atomically during a commit.

Publications. The write-buffer and persistence layer are published in [46]. [107] shows how the verification can be simplified by using the component semantics of Ch. 4 with retractions and retry in order to specify the behavior of a power failure.

Contents

12.1 Overview

Fig. 12.1 gives an overview over the components presented in this chapter. First the serialization of transactional and index nodes by the component *Node Serialization* is discussed in Sec. 12.2. The component uses the write-buffer to persist the byte representation. Several concerns must be addressed by the byte representation.

First, it must be ensured that the page-aligned partial writes, either due to I/O errors or power failures, can be detected. It is necessary to take into account the error model of flash hardware (see Ch. 8). The second aspect is that two kinds of access to the data must be provided. On the one hand, given an address of a node the component must be able to read the requested node quickly. This access pattern is used by the component *FFSC* of Ch. 11 when a file system object is read. On the other hand, for garbage collection and recovery it is

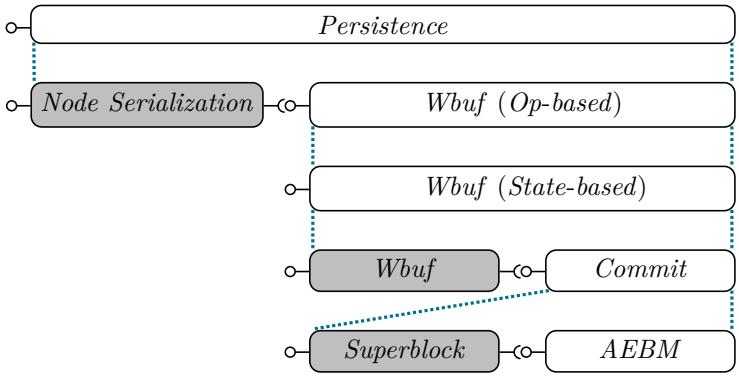

Figure 12.1: Overview over the Components responsible for Data Serialization, Buffering of Writes and Atomicity of the Commit

necessary to read all nodes and their addresses of a single erase block. The third concern is that the component must always be able to synchronize the write-buffer at the user's request. However, the write-buffer can not be filled with garbage data in order to provoke a flush, since this would prohibit reading all nodes from an erase block sequentially. The solution is to have just enough space to the next page boundary to be able to write a padding node. Padding nodes can be detected during reading and are just skipped.

The write-buffer component *Wbuf* holds a page-sized buffer and implicitly writes the data once a page boundary is crossed. In a first step the component *Wbuf* is abstracted to a specification *Wbuf* (*State-based*) with a state-based description of the effect of a power failure, i.e., the specification only employs the crash predicate and all states are synchronized. This is a normal data refinement. In the second step the state-based specification is abstracted to an operations-based specification *Wbuf* (*Op-based*) as shown in the figure. Here most of the effect of a power failure is expressed in terms of synchronized states. This step uses the crash refinement of Ch. 5.

The last implementation component *Superblock* defines the disk layout for the commit data structures and provides atomicity of the commit and atomicity for appending LEBs to the log. The first logical block is reserved for the superblock. Directly after the superblock there is space for two versions of the commit data structures. The remaining logical erase blocks are exposed to the write-buffer and node serialization components for their use. The specification and implementation is discussed in Sec. 12.4.

12.2 Persistence & Node Serialization

LEB Property Array The main data structure of the component *Node Serialization* is the *LEB Property Array* (= LPA). It is one of the commit data structures.

state *lpa*: `Array⟨LebProps⟩`

For every logical erase block it stores its current `ref-size` field and its current allocation status. A LEB is either free or in use by the journal or the index.

data type `LebProps` = `leb-props(ref-size`: ℕ, `status`: `LebStatus)`
data type `LebStatus` = `FREE` | `JOURNAL` | `INDEX`

Based on the data structure LPA, the functions `free-lebs`(*lpa*), `journal-lebs`(*lpa*) and `index-lebs`(*lpa*) are defined and return the set of logical erase block numbers with the corresponding allocation status.

In order to efficiently allocate free LEBs and find a suitable block for garbage collection a list of free LEBs *freelebs* and a binary heap *gclebs* are kept. The binary heap is represented here

as a set, in the real models this is a separate subcomponent implemented by an array-based binary heap.

state $\mathit{freelebs}\colon \texttt{List}\langle\mathbb{N}\rangle, \mathit{gclebs}\colon \texttt{Set}\langle\texttt{GCLeb}\rangle, \mathit{ramloglebs}\colon \texttt{List}\langle\mathbb{N}\rangle$

data type $\texttt{GCLeb} = \texttt{gc-leb}(\texttt{leb}\colon \mathbb{N}, \texttt{ref-size}\colon \mathbb{N})$

The binary heap is sorted by the `ref-size` field. This facilitates the implementation of the `persistence_get_gc_block` interface operation, which just removes the block with the minimal `ref-size` field from *gclebs* and returns it to the garbage collector of Ch. 11. The Invariants (freelebs-inv) and (gclebs-inv) show that the auxiliary data structures *freelebs* and *gclebs* contain the desired LEBs. Note that all LEBs of the log are already excluded from *gclebs*, since these are not eligible for garbage collection.

invariant

$$\neg\,\texttt{dups}(\mathit{freelebs}) \wedge \mathit{freelebs} = \texttt{free-lebs}(\mathit{lpa}) \qquad \text{(freelebs-inv)}$$

$$\wedge\ \mathit{gclebs} = \left\{\, \texttt{gc-leb}(l, \mathit{lpa}[l].\texttt{ref-size}) \;\middle|\; l \in \texttt{journal-lebs}(\mathit{lpa}) \setminus \mathit{ramloglebs} \,\right\} \qquad \text{(gclebs-inv)}$$

In order to quickly update the *gclebs* data structure after a commit, i.e., when the log is emptied, the component *Node Serialization* keeps the LEBs of the log *ramloglebs* separately in RAM with the additional Invariant (loglebs-lpa).

invariant $\mathit{ramloglebs} \subseteq \texttt{journal-lebs}(\mathit{lpa})$ (loglebs-lpa)

Before discussing the component *Node Serialization* further, it is necessary to introduce its subcomponent, namely the operations-based specification of the write-buffer.

Write-Buffer (Operations-based) The component *Node Serialization* interacts with the operations-based specification *Wbuf* (*Op-based*) of the write-buffer. This specification is depicted in Fig. 12.2. The component stores the block that is currently buffered (if any) in the state variable *bufleb*. Furthermore, the component exposes one volume of the erase block manager *lebs*, the current list of blocks of the log *logblocks* and the flash version of the commit data structures *froot*, *forphans* and *flpa*.

state $\mathit{bufleb}\colon \texttt{Option}\langle\mathbb{N}\rangle$

The implementation of the interface operations is omitted for brevity and most of it should already be clear from the state and the components that were already shown previously. The interface and implementation of the write-buffer is independent of the partitioning into index and journal LEBs. The component provides interface operations to move and destroy the write-buffer, which modify the state variable *bufleb* only. Buffered writing appends the new data to the current offset $\mathit{lebs}[\mathit{bufleb}.\texttt{get}].\texttt{written}$ in the buffered LEB. Note that there is no precondition that demands page alignment for the offset or for the number of bytes that are written. Reading the buffered LEB is allowed, too. The operations for mapping and unmapping of logical erase blocks of the erase block manager are exposed to the clients of the write-buffer. The commit operation now also takes the LPA, the last commit data structure, as an input parameter. Adding a LEB number to the list of LEBs of the log *logblocks* is now an explicit operation, and either performs its task atomically or returns with an error code.

Most operations are allowed in synchronized states only, i.e., if the number of bytes written to a LEB (field `written`) is aligned to one flash page, as defined by Eqn. (synchronized).

$$\texttt{synchronized}(\mathit{lebs}) \leftrightarrow \forall l < \#\,\mathit{lebs}.\ \texttt{page-aligned}(\mathit{lebs}[l].\texttt{written}) \qquad \text{(synchronized)}$$

The operations that are only allowed in synchronized states are only called right before the write-buffer is moved. At this point the state is synchronized anyway. The approach

component *Wbuf* (*Op-based*)
state $bufleb$: `Option`$\langle\mathbb{N}\rangle$, $lebs$: `Array`$\langle$`Leb`$\rangle$
$logblocks$: `List`$\langle\mathbb{N}\rangle$, $froot$: `NodeAddress`, $forphans$: `Set`$\langle\mathbb{N}\rangle$, $flpa$: `Array`$\langle$`LebProps`$\rangle$
invariant
$\#\,lebs = \#\,flpa \wedge (bufleb \neq \texttt{None} \rightarrow lebs[bufleb.\texttt{get}].\texttt{mapped?}) \wedge \forall l < \#\,lebs.\ \texttt{leb-inv}(lebs[l])$
interface operations
`awbuf_get_buf`$(; bufleb', off)$ $\{\cdots\}$
`awbuf_move_buf`(l, off)
pre $l < \#\,lebs \wedge lebs[l].\texttt{mapped?} \wedge off = lebs[l].\texttt{written} \wedge \texttt{synchronized}(lebs)$ $\{\cdots\}$
`awbuf_destroy_buf`$()$
pre `synchronized`$(lebs)$ $\{\cdots\}$
`awbuf_write_buf`$(n, buf; err)$
pre $bufleb \neq \texttt{None} \wedge lebs[l].\texttt{written} + n \leq \texttt{LEB\_SIZE} \wedge n \leq \#\,buf$ $\{\cdots\}$
`awbuf_read`$(l, off, n; buf, err)$ $\{\cdots\}$
pre $l < \#\,lebs \wedge n \leq \#\,buf \wedge off + n \leq \texttt{LEB\_SIZE} \wedge lebs[l].\texttt{mapped?}$ $\{\cdots\}$
`awbuf_map`$(l; err)$
pre $l < \#\,lebs \wedge \neg\, lebs[l].\texttt{mapped?} \wedge \texttt{synchronized}(lebs)$ $\{\cdots\}$
`awbuf_unmap`(l)
pre $l < \#\,lebs \wedge bufleb \neq \texttt{Some}(l) \wedge \texttt{synchronized}(lebs)$ $\{\cdots\}$
`awbuf_add_log_leb`$(l; err)$
pre $l < \#\,lebs \wedge lebs[l].\texttt{mapped?} \wedge \texttt{synchronized}(lebs)$
$\{\, logblocks :\!+\!= l;\ err := \texttt{ESUCCESS} \,\} \vee \{\, \texttt{fail}(; err); \,\}$
`awbuf_commit`$(flpa', froot', forphans'; err)$
pre $\#\,lebs = \#\,flpa' \wedge \texttt{synchronized}(lebs)$ $\{\cdots\}$
synchronized states
`synchronized`$(lebs)$
crash
$lebs \subseteq lebs' \wedge logblocks' = logblocks \wedge froot' = froot \wedge forphans' = forphans \wedge flpa' = flpa$
recovery
`awbuf_recover`$(; logblocks', froot', forphans', flpa', err)$ $\{\ \cdots\ \}$

Figure 12.2: Component *Wbuf* (*Op-based*)

therefore does not impose any unnecessary synchronization with flash memory. Note that the write-buffer does not feature an operation for synchronization, since synchronized states are only reached implicitly by appending the right amount of data via `awbuf_write_buf`.

With the state of the write-buffer specification the following additional invariants are expressible. The invariant states that the copy of *logblocks* kept in main memory by component *Node Serialization* is consistent and that the buffered block is either the current journal head ($logblocks.\texttt{last}$) or a block assigned to the index.

invariant

$$ramloglebs = logblocks$$
$$\wedge\ \big(\ bufleb \neq \texttt{None} \rightarrow \quad (logblocks \neq [] \wedge bufleb.\texttt{get} = logblocks.\texttt{last})$$
$$\vee\ bufleb.\texttt{get} \in \texttt{index-lebs}(lpa)\ \big)$$

Allocation of LEBs The component *Node Serialization* allocates new LEBs for the journal as depicted in Fig. 12.3 (operation `serialization_move_log_head`). If no free LEBs are available the operation returns the corresponding error code `ENOSPC`. Otherwise, a free LEB is unmapped first, in order to ensure that it is really unmapped, since previously unmapped LEBs might re-emerge after a power-failure (see the abstract specification of the erase block manager in Ch. 10). The new LEB is added to the list of LEBs of the log, i.e., added to the

component *Node Serialization*
subcomponent *Wbuf* (*State-based*)
state *lpa*: `Array`⟨`LebProps`⟩, *freelebs*: `List`⟨ℕ⟩, *gclebs*: `Set`⟨`GCLeb`⟩,
ramloglebs: `List`⟨ℕ⟩, *romode*: 𝔹
interface operations
`serialization_move_log_head`(; *err*)
if *freelebs* = [] **then**
err := `ENOSPC`
else let *l* = *freelebs*.`head` **in**
`awbuf_unmap`(*l*);
`awbuf_map`(*l*; *err*);
if *err* = `ESUCCESS` **then**
`awbuf_add_log_leb`(*l*; *err*);
if *err* = `ESUCCESS` **then**
`awbuf_move_buf`(*l*, 0);
freelebs := *freelebs*.`tail`, *lpa*[*l*].`status` := `JOURNAL`, *ramloglebs* :+= *l*;

Figure 12.3: Component *Node Serialization*: State and LEB Allocation for the Journal

state variable *logblocks*. Afterwards, the write-buffer is moved to the new LEB with an initial offset of 0 for writes, and the data structures in main memory are updated accordingly. For free LEBs it is already known that the field `ref-size` is 0 according to Invariant (refsize-free).

invariant $\forall l \in$ `free-lebs`(l). *lpa*$[l]$.`ref-size` $= 0$ (refsize-free)

For index LEBs the process is similar, specifically, the write-buffer is also moved to the newly allocated LEB in the expectation that nodes of the B$^+$-tree are written immediately afterwards.

Byte Representation of Transactional and Index Nodes As already explained the byte representation of nodes must fulfill several criteria: It must be possible 1) to detect that a page-aligned suffix of the data is missing, 2) to read all nodes in a block sequentially and distinguish them from `EMPTY` bytes, and 3) to flush the write-buffer.

In order to satisfy these criteria the byte representation of transactional and index nodes is enclosed in a header and a trailer as depicted in Fig. 12.4. Both the header and the trailer may not consist only of bytes of the value `EMPTY`. This facilitates detection of the existence of a node (header present) and of a completely written node (trailer present). The trailer is just some constant sequence of bytes `node-trailer` that is distinct from the sequence of `EMPTY` bytes and is of the fixed size `NODE_HEADER_SIZE`.

Figure 12.4: Byte Representation of Nodes

constant `node-trailer`: `Array`⟨`Byte`⟩ (node-trailer)
axiom # `node-trailer` = `NODE_HEADER_SIZE`
axiom ¬ `is-empty`(`node-trailer`, 0, `NODE_HEADER_SIZE`)

In contrast, the header is the byte representation of the algebraic data type `NodeHeader`. The byte representation is given by the serialization approach discussed in Ch. 9 with the restrictions that the byte representation never consists of bytes of value `EMPTY` only and that it is also of the fixed size `NODE_HEADER_SIZE`. The header stores whether it precedes an actual node or a *padding* node, and the size of the byte representation of the (actual or

padding) node that follows. A padding node may essentially contain arbitrary data and is used to flush the write-buffer.

data type $\texttt{NodeHeader} = \texttt{node-header}(\texttt{size}\colon \mathbb{N}, \texttt{ispadding}\colon \mathbb{B})$

For the transactional nodes of the journal and index nodes of the B$^+$-tree a serialization that is aligned to $2 \cdot \texttt{NODE\_HEADER\_SIZE}$ is chosen. This ensures that the entire byte representation shown in Fig. 12.4 is aligned to $2 \cdot \texttt{NODE\_HEADER\_SIZE}$. In the actual code the constant NODE_HEADER_SIZE is 8, 4 bytes are needed for the field size and another 4 bytes for the field ispadding of the type NodeHeader. A page of flash memory is usually either 512 bytes or 2048 bytes. It is therefore realistic to assume that Axiom (node-header-size) holds, which states that the size of a flash page is divisible by twice the size of the header of a node.

axiom $\texttt{PAGE\_SIZE} \,\%\, (2 \cdot \texttt{NODE\_HEADER\_SIZE}) = 0$ (node-header-size)

axiom $\texttt{NODE\_HEADER\_SIZE} \neq 0$

The axiom ensures that if the currently written nodes in a LEB are not aligned to a flash page, then at least $2 \cdot \texttt{NODE\_HEADER\_SIZE}$ bytes are available until the next page alignment, and it is possible for the component *Node Serialization* to write a padding node with 0 (or more) bytes. This write ensures that a synchronized state is reached and exacts a flush from the write-buffer.

The byte representation of transactional nodes and index nodes is abstracted by the predicate abs-tnodes and abs-inodes, respectively. Both abstractions work exactly the same, therefore we focus on transactional nodes in the following.

The predicate $\texttt{abs-tnodes}(\mathit{buf}, \mathit{tnodes}, \mathit{toffs}, \mathit{partial})$ states that the array *buf* contains a byte representation of the transactional nodes *tnodes* at offsets *toffs*. The flag *partial* denotes whether a failure occurred during the write of the last transactional node. If the flag is set then a partial node may be at the end of the buffer and appending additional nodes is no longer safely possible. The definition of the predicate is based on the three predicates $\texttt{abs-tnode}(\mathit{buf}, \mathit{tnode})$, $\texttt{abs-padding}(\mathit{buf})$ and $\texttt{abs-partial}(\mathit{buf})$.

The predicate $\texttt{abs-tnode}(\mathit{buf}, \mathit{tnode})$ ensures that the array *buf* contains the byte representation of the transactional node *tnode* as depicted by Fig. 12.4. Its definition is shown in Equation (abs-tnode), where the predicate serialized is an instantiation of the corresponding predicate of Ch. 9 with the restrictions as explained above.

$$
\begin{aligned}
&\texttt{abs-tnode}(\mathit{buf}, \mathit{tnode}) && \text{(abs-tnode)}\\
\leftrightarrow\ &\exists \mathit{buf}_0, \mathit{buf}_1.\quad \mathit{buf} = \mathit{buf}_0 + \mathit{buf}_1 + \texttt{node-trailer}\\
&\qquad\qquad \wedge\, \texttt{serialized}(\texttt{node-header}(\texttt{serialized-size}(\mathit{tnode}), \mathit{false}), \mathit{buf}_0)\\
&\qquad\qquad \wedge\, \texttt{serialized}(\mathit{tnode}, \mathit{buf}_1)
\end{aligned}
$$

Predicate $\texttt{abs-padding}(\mathit{buf})$ is defined similarly and asserts that the buffer contains a padding node. A partially written node satisfies $\texttt{abs-partial}(\mathit{buf})$ as defined by Eqn. (abs-partial).

$$
\begin{aligned}
&\texttt{abs-partial}(\mathit{buf}) && \text{(abs-partial)}\\
\leftrightarrow\ &\#\,\mathit{buf} \,\%\, \texttt{NODE\_HEADER\_SIZE} = 0\\
&\wedge\, \exists \mathit{buf}_0, \mathit{buf}_1.\quad \mathit{buf} = \mathit{buf}_0 + \mathit{buf}_1 \wedge \#\,\mathit{buf}_0 = \texttt{NODE\_HEADER\_SIZE}\\
&\qquad\qquad \wedge \big(\quad \texttt{is-garbage}\langle\texttt{NodeHeader}\rangle(\mathit{buf}_0)\\
&\qquad\qquad\quad \vee\, \exists \mathit{ndhd}.\ \texttt{serialized}(\mathit{ndhd}, \mathit{buf}_0) \wedge \#\,\mathit{buf}_1 \leq \mathit{ndhd}.\texttt{size}\ \big)
\end{aligned}
$$

The definition of abs-partial ensures that either the header contains garbage data, most likely because buf_0 contains only EMPTY bytes, or that the remaining buffer is to short to contain the trailer of the byte representation.

It is now possible to define the abstraction $\texttt{abs-tnodes}(\mathit{buf}, \mathit{tnodes}, \mathit{toffs}, \mathit{partial})$ of a list of transactional nodes *tnodes* and their offsets *toffs* in the buffer *buf* recursively. The idea

```
interface operations
  serialization_journal_head_append(tnodes; adrlist, err)
    if romode then err := EROFS
    let buf, bufleb, off in
      awbuf_get_buf(; bufleb, off);
      serialization_serialize_tnodes(tnodes, bufleb.get, off; buf, adrlist, err);
      if err = ESUCCESS then
        awbuf_write_buf(# buf, buf; err);
        if err ≠ ESUCCESS then romode := true;   // Enter read-only mode
  serialization_read_tnode(adr; tnode, err)
    let buf = Array⟨Byte⟩(adr.size) in
      awbuf_read(adr.leb, adr.off, adr.size; buf, err);
      if err = ESUCCESS then
        serialization_deserialize_tnode(adr.size, buf; tnode, err);
```

Figure 12.5: Component *Node Serialization*: Appending Nodes to the Journal and Reading Nodes

is to break *buf* into a first part buf_0, which satisfies either `abs-tnode`$(buf_0, tnodes.\texttt{head})$ or `abs-padding`(buf_0), and a rest buf_1 that contains the byte representation of the remaining nodes and offsets. If *partial* is *true*, then the last part of the buffer may satisfy `abs-partial`. The details are omitted for brevity.

Appending Nodes Fig. 12.5 shows how transactional nodes are appended to the journal. The byte representation is performed by the operation `serialization_serialize_tnodes`. Its implementation is not shown, since it just serializes each node as shown in Fig. 12.4 on page 155. The LEB *bufleb*.`get` for the nodes and the current offset *off* is needed in order to calculate the address for each node. The addresses are returned in the variable *adrlist*. Afterwards, the buffer *buf* is appended to the journal head. Before the operation

$$\texttt{abs-tnodes}(lebs[bufleb.\texttt{get}].\texttt{data}, tnodes_0, toffs_0, false)$$

holds for some transaction nodes $tnodes_0$ and some offsets $toffs_0$. If appending succeeds, then

$$\texttt{abs-tnodes}(lebs[bufleb.\texttt{get}].\texttt{data} + buf, tnodes_0 + tnodes, toffs_0 + toffs, false)$$

is shown afterwards, where *buf* contains the serialized data of all transactional nodes *tnode*. Note the last parameter *false*, which indicates that appends are still possible afterwards. If appending fails on the other hand, the file system enters read-only mode and

$$\texttt{abs-tnodes}(lebs[bufleb.\texttt{get}].\texttt{data} + buf_1, tnodes_0 + tnodes_1, toffs_0 + toffs_1, true)$$

is shown for some prefix buf_1 of *buf* that is aligned to `NODE_HEADER_SIZE` and some prefix $tnodes_1$ and $toffs_1$ of *tnodes* and *toffs*, respectively. Appending additional nodes afterwards is no longer allowed, as indicated by the partial flag that is now set to *true*.

Reading Nodes A single transactional node is read as shown in Fig. 12.5. First, enough space is allocated for the read access. Note that here the field `size` of the address is used. If addresses did not carry the additional size field, several I/O operations would be needed to read the header and only then the remainder of the node. Afterwards, the respective part of the LEB is read and the byte representation is decoded. In order to read the transactional nodes of an entire LEB, which is needed by the garbage collection and by the replay of the index after a power failure, scanning of the whole block is needed. The details are omitted here, since the operation is quite intricate and has to check that there are actually nodes and

interface operations

```
serialization_sync(; err)
  if romode then err := EROFS
  else let bufleb, off in
    awbuf_get_buf(; bufleb, off);
    if bufleb ≠ None ∧ ¬page-aligned(off) then
      let buf = Array⟨Byte⟩(PAGE_SIZE − (off % PAGE_SIZE)) in
        serialization_padding_node(; buf, err);
        if err = ESUCCESS then
          awbuf_write_buf(# buf, buf; err);
          if err ≠ ESUCCESS then romode := true;   // Enter read-only mode
serialization_commit(froot′, forphans′; err)
  awbuf_destroy_buf();
  awbuf_commit(lpa, froot′, forphans′; err);
  if err = ESUCCESS then
    ...   // Add LEBs of ramloglebs to gclebs
    ramloglebs := [];
    ...   // Unmap all LEBs l with lpa[l].ref-size = 0
```

Figure 12.6: Component *Node Serialization*: Synchronization & Commit

not just garbage data or EMPTY bytes. Essentially, the procedure has to reconstruct *tnodes* and *toffs* given that for the corresponding LEB l the formula

$$\texttt{abs-tnodes}(lebs[l].\texttt{data}, tnodes, toffs, partial)$$

holds for some value of the flag *partial*. Padding nodes need to be skipped as well.

Synchronization Fig. 12.6 shows how synchronization is achieved. If the current offset of the write-buffer is not aligned, then a buffer *buf* for a padding node is allocated. The padding node spans the space to the next page boundary. The header and trailer of the padding node are written to *buf* and the buffer is appended to the currently buffered logical erase block. The abstraction

$$\texttt{abs-tnodes}(lebs[bufleb.\texttt{get}].\texttt{data}, tnodes, toffs, false)$$

ensures that there is enough space for the padding node. After a successful write of the padding node,

$$\texttt{abs-tnodes}(lebs[bufleb.\texttt{get}].\texttt{data} + buf, tnodes, toffs, false)$$

is established, where *buf* contains the complete byte representation of the padding node. If the write fails then

$$\texttt{abs-tnodes}(lebs[bufleb.\texttt{get}].\texttt{data} + buf_0, tnodes, toffs, true)$$

holds, where buf_0 is again a prefix of *buf* that is aligned to NODE_HEADER_SIZE.

Commit Fig. 12.6 also depicts the commit operation of the component *Node Serialization*. First, the buffered block *bufleb* is reset to None by a call to awbuf_destroy_buf. Then a commit of the subcomponent *Wbuf* (*Op-based*) is triggered with the current LEB Properties Array *lpa* and the flash orphans *forphans′* and the root of the flash index *froot′*. The latter two parameters are passed on from the *FFSC* and B^+-*tree* components. After the commit, the log is empty. All LEBs that were part of the log prior to the commit need to be added to the LEBs available for garbage collection, i.e., to the *gclebs* data structure. Finally, all LEBs

```
recovery
  serialization_recover(; logblocks′, forphans′, froot′, err)
    awbuf_recover(; logblocks′, froot′, forphans′, lpa, err);
    if err = ESUCCESS then
      ...   // Mark all LEBs in logblocks′ as allocated for the journal
      ...   // Unmap all LEBs l with lpa[l].ref-size = 0
      ...   // Initialize gclebs and freelebs
      ramloglebs := logblocks′, romode := false;
```

Figure 12.7: Component *Node Serialization*: Recovery

that have a field `ref-size` of 0 are deallocated and unmapped. This is a form of garbage collection of the journal and index LEBs, and is especially necessary for the index LEBs. Otherwise, the flash index will incrementally use up all LEBs of the device. With this measure we get Thm. 11.

Theorem 11 (No Unused Nodes After Commit)**.** *After the commit of the file system, no unused LEBs are mapped.*

Proof. As shown by Fig. 12.6 the commit operation deallocates all LEBs with a field `ref-size` of 0. By Invariant (refsize-inv-rindex) on page 138 and (refsize-inv-btree) on page 147 all other LEBs are still referenced by the RAM index and contain live transactional nodes or are referenced by the B^+-tree and therefore contain live nodes of the flash index. □

Recovery Fig. 12.7 depicts the recovery of the component *Node Serialization*. The recovery of the write-buffer returns the commit data structures. The component *Node Serialization* then marks all LEBs in the log as allocated for the journal, i.e., for all $l \in logblocks'$ the field $lpa[l]$.`status` is set to `JOURNAL`. Then all unreferenced LEBs are deallocated (analogous to `serialization_commit` in Fig. 12.6) and the other state variables of the component are initialized appropriately.

Abstraction Relation The essential part of the abstraction relation between the component *Node Serialization* and its specification *Persistence* (see Fig. 11.15 on page 148) is the correspondence between the transactional node stores *tns*, its commit version *ftns*, invalid LEBs *invallebs*, the index node stores *ins* and its commit version *fins* on the one hand, and the logical erase blocks *lebs* and the LEB Property Arrays *lpa* and *flpa* on the other hand.

For the transactional node store *tns* and invalid LEBs *invallebs* the abstraction relation (abs-tnode-store) is used. The abstraction relation states that exactly the LEBs that are allocated for the journal, i.e., have the status `JOURNAL` in the LPA, are in the domain of the transactional node store *tns*. The `ref-size` field in the *lpa* always matches the corresponding field in *tns*. However, only if the LEB l is valid, i.e., if $l \notin invallebs$ holds, then the bytes in the LEB l correspond to the transactional nodes and offsets of the transactional node store *tns*.

abstraction relation (abs-tnode-store)

$$
\begin{aligned}
& dom(tns) = \texttt{journal-lebs}(lpa) \\
\wedge\ & \big(\forall l \in tns.\ tns[l].\texttt{ref-size} = lpa[l].\texttt{ref-size} \big) \\
\wedge\ & \big(\forall l \in tns \setminus invallebs.\ \texttt{abs-leb}(l, lebs[l], tns[l].\texttt{tnodes}, tns[l].\texttt{toffs}, bufleb, romode) \big)
\end{aligned}
$$

where

$$
\begin{aligned}
& \texttt{abs-leb}(l, leb, tnodes, toffs, bufleb, romode) \\
\leftrightarrow\ & leb.\texttt{mapped?} \\
& \wedge\ \texttt{abs-tnodes}(leb.\texttt{data}[0..leb.\texttt{written}], tnodes, toffs, bufleb \neq \texttt{Some}(l) \vee romode)
\end{aligned}
$$

The abstraction for individual LEBs `abs-leb` splits the data into the first part where the encoded data lies and a remainder that only contains bytes with the value `EMPTY`.[1] Note that the abstraction only allows for appending of nodes, i.e., the last parameter *partial* of `abs-tnodes` is true, if the LEB is buffered and the component is not in read-only mode.

The abstraction between the index node store *ins*, the LEB Properties Array *lpa* and the logical erase blocks *lebs* is analogous to (abs-tnode-store).

For the commit version of the transactional node store *ftns* and the commit version of the LEB Property Array *flpa* the abstraction relation (abs-tnode-store-commit) is used. Only the journal LEBs with a nonzero `ref-size` field are part of *ftns*, since the operation `serialization_commit` deallocates and unmaps these LEBs only after persisting the LPA as shown in Fig. 12.6. For this reason the recovery operation (see Fig. 12.7) also has to deallocate and unmap these LEBs after loading the flash version of the LPA. Only then is the abstraction relation (abs-tnode-store) reestablished after a power failure.

abstraction relation (abs-tnode-store-commit)

$$\forall l < \#\,\mathit{flpa}.\quad \big(\ l \in \mathit{ftns} \leftrightarrow l \in \texttt{journal-lebs}(\mathit{flpa}) \wedge \mathit{flpa}[l].\texttt{ref-size} \neq 0\ \big) \\ \wedge\ \big(\ l \in \mathit{ftns} \rightarrow \mathit{ftns}[l].\texttt{ref-size} = \mathit{flpa}[l].\texttt{ref-size}\ \big)$$

The abstraction relation for the commit version of the index node store *fins* and *flpa* is again analogous.

Additionally, data refinement by Thm. 3 on page 59 must preserve synchronized states. The refinement between *Persistence* and *Node Serialization* is the only refinement where not only the artificial synchronized flag *synced* is considered, but the actual synchronization predicate of the write-buffer. Abstraction relation (abs-synchronized) ensures that synchronized states are preserved correctly, where *synced* is the state variable of the specification component *Persistence*.

abstraction relation $\mathit{synced} \rightarrow \texttt{synchronized}(\mathit{lebs})$ (abs-synchronized)

Finally, the state of the buffer *wbufstate* of *Persistence* must be related to the write-buffer. This is rather trivial and omitted. We conclude with the correctness and crash-safety of the component *Node Serialization*.

Theorem 12 (Correctness & Atomicity of Node Serialization)**.** *The component Persistence is refined by the component Node Serialization.* □

The main difficulties during the invariant and refinement proofs are 1) handling the byte representation of nodes and 2) again the details of the commit with garbage collection of index LEBs and recovery.

12.3 Write-Buffer: State-Based vs. Operations-Based

The state of the component *Wbuf* consists of the LEB number *bufleb* that is being cached and the write-buffer *wbuf* itself. Fig. 12.8 depicts the component structure and its state. The write-buffer *wbuf* stores the a page-sized array `data`, the current offset `off` of the write-buffer in the LEB and the number of bytes `written` currently cached in the write-buffer.

data type $\texttt{WriteBuffer} = \texttt{wbuf}(\texttt{data}\colon \texttt{Array}\langle\texttt{Byte}\rangle, \texttt{off}\colon \mathbb{N}, \texttt{written}\colon \mathbb{N})$

Invariants Invariant (wbuf-inv) states that the write-buffer has the size of a flash page. Furthermore, the write-buffer is never completely filled, since then a page-aligned write is possible, i.e., the number of bytes written to the buffer is always below the size of a flash

[1] The remainder only contains `EMPTY` bytes due to Invariant (leb-inv) on page 95, which also holds for the subcomponent *Wbuf* (*Op-based*).

```
component Wbuf
  subcomponent Commit                                   (Fig. 12.9 on page 164)
state bufleb: Option⟨ℕ⟩, wbuf: WriteBuffer, romode: 𝔹
invariant wbuf-inv(bufleb, wbuf, lebs)
interface operations
  wbuf_write_buf(n, buf; err)
    if romode then err := EROFS;
    else if wbuf.written + n ≥ PAGE_SIZE then
      let nwrite = align↓(wbuf.written + n, PAGE_SIZE) in
      let nbuf = nwrite − wbuf.written in
      let nrest = (wbuf.written + n) % PAGE_SIZE in
      let buf_0 = Array⟨Byte⟩(nwrite) in
        buf_0 := copy(wbuf.data, 0, buf_0, 0, wbuf.written);
        buf_0 := copy(buf, 0, buf_0, wbuf.written, nbuf);
        commit_write(bufleb.get, wbuf.off, 0, nwrite, buf_0; err);
        if err = ESUCCESS then
          wbuf.data := copy(buf, nbuf, wbuf.data, 0, nrest);
          wbuf.off :+= nwrite;
          wbuf.written := nrest;
        else
          romode := true;
    else
      wbuf.data := copy(buf, 0, wbuf.data, wbuf.written, n);
      wbuf.written :+= n;
      err := ESUCCESS;
```

Figure 12.8: Component *Wbuf*

page. Finally, if the component is not in read-only mode, then the offset *wbuf*.off stored in main memory corresponds to the current number of bytes written to the buffered LEB, i.e., it corresponds to *lebs*[*bufleb*.get].written. This ensures that component actually writes the data sequentially.

invariant wbuf-inv(*bufleb*, *wbuf*, *lebs*) (wbuf-inv)

where

$$
\begin{aligned}
&\texttt{wbuf-inv}(bufleb, wbuf, lebs)\\
\leftrightarrow\quad &\#\, wbuf.\texttt{data} = \texttt{PAGE\_SIZE} \wedge wbuf.\texttt{written} < \texttt{PAGE\_SIZE}\\
&\wedge \big(\ bufleb \neq \texttt{None} \rightarrow \quad bufleb.\texttt{get} < \#\, lebs \wedge lebs[bufleb.\texttt{get}].\texttt{mapped?}\\
&\qquad\qquad\qquad\qquad \wedge (\neg\, romode \rightarrow lebs[bufleb.\texttt{get}].\texttt{written} = wbuf.\texttt{off})\big)
\end{aligned}
$$

Buffered Writing The main operations of the component *Wbuf* is reading and writing. All other operations just modify the data structures in main memory accordingly, or are passed through to the subcomponent *Commit*. Fig. 12.8 shows the operation wbuf_write. If a page-aligned write is possible, then the component assembles a new buffer buf_0 and writes all *nwrite* bytes in a single I/O operation. The remaining *nrest* bytes are copied into the write-buffer *wbuf* afterwards and the offset of the buffer is shifted accordingly. In case of a write failure a switch to read-only mode is performed. If no page-aligned write is possible, then the bytes of the input buffer *buf* are just cached by appending them to the write-buffer *wbuf*.

The read operation essentially just reads the requested LEB. If the LEB is buffered, then the read operation potentially needs to copy the corresponding part of the write-buffer to the output buffer.

State-Based Specification The state-based specification *Wbuf* (*State-based*) of the write-buffer is the same as the operations-based specification seen in Fig. 12.2 on page 154 only the synchronized and crash predicate are given by (wbuf-crash-state-based). All states of component *Wbuf* (*State-based*) are synchronized and according to the crash predicate a partially written page of every LEB is removed by a power failure. This corresponds to the effect of loosing the write-buffer during a power failure.

synchronized states (wbuf-crash-state-based)

$$true$$

crash

$$lebs' \downarrow lebs \wedge logblocks' = logblocks \wedge froot' = froot \wedge forphans' = forphans \wedge flpa' = flpa$$

where

$$lebs' \downarrow lebs \leftrightarrow \#\, lebs' = \#\, lebs \wedge \forall l < \#\, lebs.\ lebs'[l] \downarrow lebs[l]$$
$$leb' \downarrow leb \leftrightarrow \big(leb.\texttt{mapped}? \rightarrow leb' = \texttt{mapped}(leb.\texttt{data}\downarrow)\big) \wedge \big(leb.\texttt{erased}? \rightarrow leb' = leb\big)$$
$$buf\downarrow \equiv buf[0..\texttt{align}\downarrow(\#\, buf, \texttt{PAGE\_SIZE})]$$

Abstraction Relation With the abstraction relation (abs-wbuf) a data refinement between the component *Wbuf* and the state-based specification *Wbuf* (*State-based*) is proven, where $lebs_a$ and $lebs_c$ denote the corresponding state variables of component *Commit* and *Wbuf* (*State-based*), respectively.[2] The abstraction relation essentially copies the write-buffer over the buffered LEB.

abstraction relation (abs-wbuf)

$$\begin{aligned}&\#\, lebs_a = \#\, lebs_c\\ \wedge\, &\forall l < \#\, lebs_c.\quad \big(bufleb \neq \texttt{Some}(l) \rightarrow lebs_a[l] = lebs_c[l]\big)\\ &\qquad \wedge \big(bufleb = \texttt{Some}(l) \rightarrow lebs_a[l] = \texttt{mapped}(\texttt{copy}(wbuf.\texttt{data}, 0,\\ &\qquad\qquad lebs_c[l].\texttt{data}, wbuf.\texttt{off},\\ &\qquad\qquad wbuf.\texttt{written}),\\ &\qquad\qquad lebs_c[l].\texttt{written} + wbuf.\texttt{written})\big)\end{aligned}$$

Thm. 13 shows that the write-buffer is correct and crash-safe with respect to the state-based specification.

Theorem 13 (Correctness & Crash-Safety of the Write-Buffer)**.** *The component Wbuf refines the component Wbuf* (*State-based*)*.* □

Operations-Based Specification As already shown in Fig. 12.2 on page 154, the synchronized and crash predicate of the operations-based specification of the write-buffer are given by (wbuf-crash-operations-based). The definition of $\texttt{synchronized}(lebs)$ is given by Equation (synchronized) on page 153 and states that in synchronized states all LEBs are written up to a page-aligned offset.[3] The domain of the crash predicate encompasses only the

[2] Note that some details about the write-buffer in read-only mode are omitted here for clarity.
[3] The formula $lebs \subseteq lebs'$ is defined in Sec. 10.1.

synchronized states, i.e., the operations-based specification considers crashes only in states where no pages are written partially.

synchronized states (wbuf-crash-operations-based)

$\texttt{synchronized}(lebs)$

crash

$\texttt{synchronized}(lebs) \wedge lebs \subseteq lebs'$

$\wedge\ logblocks' = logblocks \wedge froot' = froot \wedge forphans' = forphans \wedge flpa' = flpa$

A crash refinement between both specification by Thm. 4 on page 61 shows that we can switch from the state-based specification to the operations-based specification. This removes the effect of the write-buffer during a power failure completely from the crash predicate and instead employs the synchronized states as a specification mechanism. This facilitates the implicit propagation of the effect upwards all other data refinements until the top-level specification *POSIX* is reached.

Theorem 14 (State-Based & Operations-Based Interpretation of the Write-Buffer)**.** *The component Wbuf (State-based) is a crash refinement of the component Wbuf (Op-based).*

Proof. First of all, note that most operations of the components are only allowed in synchronized states and that these operations always yield synchronized states. Therefore, finding a retraction or re-execution is limited to the `awbuf_get_buf`, `awbuf_read` and `awbuf_write_buf`. The first two operations are obviously retractable, since they do not modify the persistent state.

The implementation of `awbuf_write_buf` is essentially the same as `aebm_write` in Fig. 10.2 on page 94, just that the LEB and the offset are implicitly given by the write-buffer. All executions of `awbuf_write_buf` that did not cross the last page boundary are retractable. All other executions are completable in such a way that they write exactly up to the page boundary and yield a synchronized state. A visualization of the situation is already given by Fig. 6.2 on page 70 where write W5 is retracted and write W4 is re-executed up to the respective page boundary. □

12.4 Superblock, Commit Atomicity & Flash Layout

The remaining gap between the write-buffer of the previous section and the erase block manager of Ch. 10 is filled by the component *Superblock*. Note that several invariants are omitted for brevity and only described informally. Fig. 12.9 depicts its specification component *Commit*.

Specification (Component *Commit*) It exposes only one volume to its clients captured by the state variable *lebs*. The other state variables correspond to the version of the commit data structures stored at the point of the last successful commit. The specification exposes a commit operation, which if successful persists the commit data structures. If unsuccessful no change in the state is visible, i.e., the commit data structures remain intact. Furthermore, logical erase block number can be appended to the log. Here again, atomicity is crucial. A power failure leaves the commit data structures and the log unchanged. Only unmapped LEBs may re-emerge, which is essentially the behavior of the erase block manager of Ch. 10.

The remainder of this section explains the implementation of the specification *Commit*.

component *Commit*
state *lebs*: `Array⟨Leb⟩`, *logblocks*: `List⟨ℕ⟩`,
froot: `NodeAddress`, *forphans*: `Set⟨ℕ⟩`, *flpa*: `Array⟨LebProps⟩`
invariant
$\#\,lebs = \#\,flpa \wedge \forall l < \#\,lebs.$ `leb-inv`$(lebs[l])$
interface operations
`commit_commit`$(froot', forphans', flpa'; err)$
$\{\ logblocks := [],\ froot := froot',\ forphans := forphans',\ flpa := flpa',\ err :=$ `ESUCCESS` $\}$
$\vee\ \{$ `fail`$(;\ err)\ \}$
`commit_is_commit_required`$(;\ doCommit)$
$doCommit := ?$
`commit_log_add_leb`$(l; err)$
$\{\ logblocks :+= l; err :=$ `ESUCCESS` $\} \vee \{$ `fail`$(;\ err)\ \}$
// operations for reading, writing and (un)mapping of LEBs of component *AEBM*
crash
$lebs \subseteq lebs' \wedge logblocks' = logblocks \wedge froot' = froot \wedge forphans' = forphans \wedge flpa' = flpa$
recovery
`commit_recover`$(; logblocks', froot', forphans', flpa', err)\ \{\ \cdots\ \}$

Figure 12.9: Specification Component *Commit*: The component provides an operation to reliably and atomically persist the commit data structures *froot*, *forphans* and *flpa* and recover them after a power failure. The other important operation is appending new LEBs to the log atomically.

Flash Layout Fig. 12.10 depicts the layout that the component *Superblock* imposes on the flash device. Note that here the rectangles denote entire logical erase block, not just a single page of flash memory. The superblock is stored in the first logical erase block of the volume. For each of the commit data structures space for two versions is provisioned. The superblock stores the LEB numbers of the current location of the list of LEBs of the log and of each of the data structures, except for the flash address of the root node of the B^+-tree of Ch. 11, which is directly kept in the superblock data structure. Additionally, the superblock stores the LEB number of the first erase block that is usable by the client. These pointers are depicted as **solid red arrows**.

The component keeps the current version of the superblock in main memory for easy access to the LEB numbers.

state *sb*: `Superblock`
data type `Superblock` = `superblock(froot: NodeAddress, logleb: ℕ, orphansleb: ℕ, lpaleb: ℕ, mainleb: ℕ)`

The field `mainleb` denotes the number of the first LEB exposed to the write-buffer component. The component uses a constant default volume identifier `v` as its volume for the interaction with the erase block manager. For the superblock and for the flash orphans an encoding that is aligned to the size of a flash page is chosen as discussed in Ch. 9. The LEB of the superblock is always mapped according to Invariant (superblock-inv), which also ensures that its data is unchanged by a power failure of the erase block manager.

invariant (superblock-inv)
$avols[\mathtt{v}][0].\mathtt{mapped}? \wedge \mathtt{serialized}\langle\mathtt{Superblock}\rangle(sb, avols[\mathtt{v}][0].\mathtt{data})$
$\wedge\ avols[\mathtt{v}][sb.\mathtt{logleb}].\mathtt{mapped}?$
$\wedge\ \big(\exists\, logblocks.\ \mathtt{serialized}\langle\mathtt{List}\langle\mathbb{N}\rangle\rangle(logblocks, avols[\mathtt{v}][sb.\mathtt{logleb}].\mathtt{data})\big)$
$\wedge\quad \cdots$

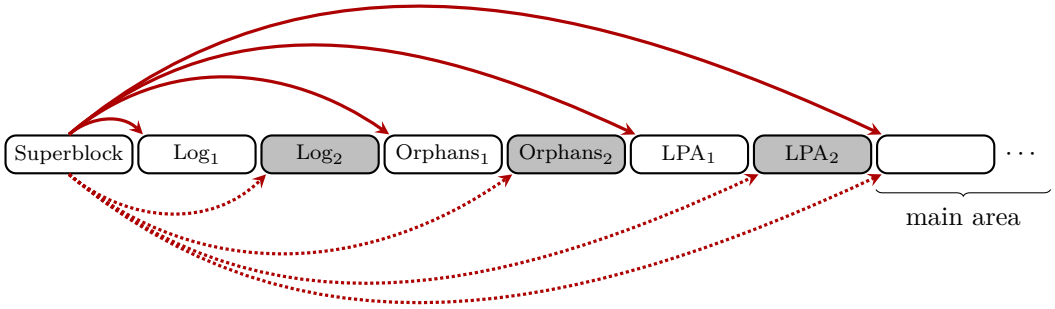

Figure 12.10: Component *Superblock*: Flash Layout of the Superblock and Commit Data Structures. Solid red arrows denote the current state and dotted red arrows the state after the next commit. Gray erase blocks are currently not mapped.

The LEBs of the log are serialized explicitly, because the byte representation of each entry must match an entire flash page exactly, in order to be able to append one entry at a time atomically to the log, and may not just contain bytes with the value `EMPTY`.[4] The LPA is also serialized explicitly in order to guarantee a fixed size. A complicating factor for the LPA is that multiple LEBs are necessary to store it.

An additional invariant (omitted here) asserts that the LEB numbers stored in the superblock *sb* used for the serialized data structures correspond to either one of their possible locations shown in Fig. 12.10. The field `mainleb` must be calculated, because the size of the LPA depends on the size of the volume *volsize* that is exposed to the client. Initially, the client of the file system can choose *volsize*.

invariant $sb.\texttt{mainleb} = 5 + 2 \cdot \texttt{lpalebs}(volsize)$ (main-area-inv)

The function `lpalebs` in Invariant (main-area-inv) calculates the number of LEBs necessary to store a LEB Properties Array *lpa* with $\#\, lpa = volsize$.

LEBs of the Log For the LEB that holds the log an additional offset *logoff* is kept in main memory with invariant (logoff-inv) that the offset corresponds to the number of bytes already written to the LEB.

state $logoff : \mathbb{N}$

invariant $logoff \neq \texttt{LEB\_SIZE} \rightarrow avols[\texttt{v}][sb.\texttt{logleb}].\texttt{written} = logoff$ (logoff-inv)

Fig. 12.11 depicts the implementation of appending a LEB number l to the log. If the write fails or if the space is exhausted, i.e., once $logoff = \texttt{LEB\_SIZE}$ holds the component signals to its client that a commit is required. Clients can also proactively check that there is enough space for another entry in the log. Appending entries to the log is atomic, because the error model of the flash hardware of Ch. 8 either completely writes a page or does not write the page at all.

Commit & Recovery The commit operation is depicted schematically Fig. 12.11. First the new superblock is calculated, i.e., the pointer of the log and the commit data structures is moved to their other location denoted by the function `switch`. Afterwards, the LEBs that correspond to this location are unmapped and mapped anew, since they may have reemerged

[4] Note that this could be improved upon by allocating more than one LEB for the journal at once and using them in sequence.

```
component Superblock
  subcomponent AEBM                                          (Fig. 10.2 on page 94)
state sb: Superblock, logoff: ℕ
invariant (superblock-inv) ∧ (logoff-inv) ∧ (main-area-inv)
interface operations
  superblock_log_add_leb(l; err)
    if LEB_SIZE < logoff + PAGE_SIZE then err := ECOMMIT
    else let buf = Array⟨Byte⟩(PAGE_SIZE), size then
      serialize⟨T⟩(ℕ, 0; buf, size, err);
      if err = ESUCCESS then
        aebm_write(v, sb.logleb, logoff, 0, PAGE_SIZE, buf; err);
        if err = ESUCCESS then logoff :+= PAGE_SIZE else logoff := PAGE_SIZE
  superblock_is_commit_required(; doCommit)
    doCommit := logoff = LEB_SIZE
  superblock_commit(froot′, forphans′, flpa′; err)
    let sb′ = switch(sb, froot′, volsize), buf in
      ...    // Unmap and map the LEBs for the new log and for lpa′ and forphans′
      ...    // Write lpa′ and forphans′
      ...    // Serialize sb′ to buf
      aebm_change(v, 0, # buf, buf; err);      // Unmap other LEBs
```

Figure 12.11: Component *Superblock*: Appending LEBs to the Log and Commit (error-handling omitted for commit)

after a power failure due to the crash behavior of the erase block manager. This ensures that the new log is empty and all other data structures are writable. In the next step the new versions of the flash orphans $forphans'$ and the LPA lpa' are written. Afterwards, the superblock is exchanged *atomically* with the help of the erase block manager's operation `aebm_change`. Finally, the LEBs holding the old versions of the data structures may be unmapped and are thereby free for reuse to the erase block manager.

The recovery from a power failure needs to read the superblock and afterwards the current version of the log, the flash orphans and the LPA. These data structures are then returned to the client components' recovery operations.

All other operations that access logical erase block l just call the corresponding operation of the subcomponent *AEBM* with the volume identifier `v` and the LEB $sb.\texttt{mainleb} + l$.

Theorem 15 (Correctness & Crash-Safety of Commit)**.** *The component Superblock refines the specification Commit.*

Proof. The abstraction relation between the component *Superblock* with its subcomponent *AEBM* and the specification component *Commit* essentially states that the values for the existential quantifiers of Invariant (superblock-inv) correspond to the state variables of the component *Commit* of the same name.

abstraction relation (superblock-abs)

$$\begin{aligned} &\#\, lebs + sb.\texttt{mainleb} = \#\, avols[\texttt{v}] \wedge lebs = avols[\texttt{v}][sb.\texttt{mainleb}..] \\ \wedge\ &\texttt{serialized}\langle\texttt{List}\langle\mathbb{N}\rangle\rangle(logblocks, avols[\texttt{v}][sb.\texttt{logleb}].\texttt{data}) \\ \wedge\ &\quad \ldots \end{aligned}$$

The abstraction relation (superblock-abs) additionally ensures that the contents of the volume after the LEB $sb.\texttt{mainleb}$ are exposed to the client. Note that the behavior during a power failure just propagates upwards from component *AEBM* to component *Commit*. The main

difficulty during the invariant and refinement proofs is ensuring that changes to one data structure do not invalidate any other data structure.

For the refinement proof of commit it has to be shown that failure leads to no change in the abstract state. This is guaranteed since all writes before the `aebm_change` operation on the superblock are performed on the non-active version of the data structures. If the exchange of the superblock fails, the old version of the superblock and all other data structures are still intact. Otherwise, the new superblock is written correctly and the new versions are visible to the client. □

12.5 Related Work

The file system Bilbyfs [77, 11, 9] uses a write-buffer similar to the one considered in this thesis. Bilbyfs ensures that only sequential writes are performed by the file system. The verification of crash-safety and the correctness of recovery still remains future work. It is, however, proven that `iget` and `sync` are functionally correct with respect to a specification of AFS similar to the one of Flashix. In order to specify the correctness of write-back caching on the level of AFS, Amani [9] stores a list of higher-order state transformers that capture each of the AFS operations explicitly.

The verified file system FSCQ [27, 28, 26] features a journal that caches writes across several POSIX operations, too. Their model of a hard disk is more general than our model of flash hardware with respect to power failures. It allows the magnetic disk to asynchronously write sectors and to re-order the writes. The effect of these caches is modeled explicitly in FSCQ, too.

Flashix employs the write-buffer primarily to cope with the limitations of flash hardware. The write-buffer caches until a page-aligned write is possible, i.e., the purpose of the write-buffer in Flashix is not necessarily to achieve better performance. In order to improve the performance, the write-back caches of VFS should be used in the future by Flashix. These write-back caches are, however, not in the scope of this thesis, but are currently being modeled and proven correct. The operations-based view of the write-buffer in Flashix has the advantage that it propagates upwards the entire refinement hierarchy implicitly and that the specification mechanism is easier to understand and to validate than an explicit, state-based specification of a crash.

FSCQ uses two allocators, one for disk blocks and one for inodes, which are stored on the hard disk. This is similar to the LPA data structure of Flashix. However, it is possible to overwrite the on-disk version of the data structure when modifications are performed. In Flashix these data structures are write-back cached due to the limitations of flash hardware.

Part II

Verification of Lock-Based Concurrency in File Systems

13 Concurrent, Crash-Aware Components & Refinement

Summary. This chapter first revisits Lipton's theory of reductions on a simple lock-based example. Lipton reductions facilitate composing atomic blocks by proving commutations about individual program parts of a concurrent system. The resulting system has the same observable and termination behavior as the original system. This approach integrates well with atomicity refinement of sequential components of Ch. 5, where only atomicity with respect to power failures is considered. This chapter extends atomicity refinement to non-retracting, concurrent components by leveraging Lipton reductions. In practice, most of the commutations are inferred automatically by providing a set of ownership annotations and proving that the ownership discipline is adhered to. An ownership annotation for (part of) a data structure essentially states, which locks must be held in order to access it. After a component is abstracted with an atomicity refinement based on ownership annotations, it consists mostly of atomic blocks as shown for the erase block manager in Ch. 14. The resulting system has stronger invariants and for the remaining non-atomic operations and the recovery operation a direct proof of linearizability with respect to an abstract specification becomes feasible.

Publications. This chapter extends our previous work on crashes [107, 47] by integrating atomicity refinement of Ch. 5 with Lipton reductions.

Contents

13.1 Approach

The general approach for the development of a concurrent, crash-aware system is similar to that of a sequential, crash-aware system as shown by Fig. 13.1.

We start out with a concurrent component C. We incrementally increase the atomicity with respect to power failures and concurrency via an atomicity refinement integrated with Lipton reductions. This yields a component C'. Opportunities for Lipton reductions are inferred mostly automatically based on annotations of ownership. The annotations lead to

assertions in the program that are proven by a rely/guarantee as shown in Sec. 4.6. The rely/guarantee proof also establishes divergence-freedom of the component. Divergence-freedom is a necessary condition in order to increase the atomicity with respect to power failures, because crash-introducibility of Ch. 5 requires the existence of a terminating execution with a final state that subsumes the crash behavior of an intermediate state.

Afterwards, a proof of linearizability, or, in the case where all operations are already atomic, data refinement becomes possible, because the invariants are now strong enough to prove a commuting diagram for the reset transition of the component.

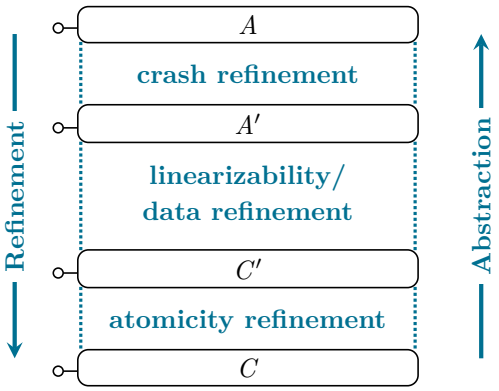

Figure 13.1: Refinement Approach for Concurrent, Crash-Aware Components

The implementation of ownership annotations, Lipton reductions and the inference of atomic sections was implemented as part of this thesis in the interactive proof assistant KIV and constitutes part of the contribution of this thesis.

This chapter is structured as follows. Sec. 13.2 first provides an example of a concurrent component and applies standard Lipton reductions to it. Sec. 13.3 extends Lipton reductions with additional proof obligations ensuring that atomicity with respect to power failures is also guaranteed. Sec. 13.4 shows how ownership annotations gives opportunities for applying Lipton reductions automatically for lock-based concurrency. Annotations of ownership are also employed to provide synchronization across the boundaries of components. This decreases the need for additional locks in subcomponents, when the client component already takes care of synchronization.

13.2 A Concurrent Counter & Lipton Reductions

Fig. 13.2 shows how mutexes and reader/writer locks are modeled with the help of atomic blocks. The mutex data structure itself stores the thread identifier of the thread owning the mutex. This facilitates the expression of relies and invariants for the verification. The figure shows the standard interface of mutexes with a procedure for locking and one for unlocking. The lock procedure waits until the mutex is free and then locks it atomically. For unlocking we have to ensure that the mutex is actually locked by the current thread and afterwards the mutex is set to free. The assertion ensures compatibility with the POSIX standard [3]. Reader/writer locks are specified similarly in the figure with the additional function `readers`, which returns the set of threads that have acquired read access, and includes a writer if one is present. The predicate _.`readers`? returns true if the reader/writer lock is constructed by the constructor `readers`. A writer has exclusive access, i.e., the thread may always read and write.

Fig. 13.4 shows a concurrent counter (left-hand side) implemented as a concurrent component. The increment operations returns the new value of the counter to the caller. The operation is protected from interference of other threads with a simple mutex. The thread identifier is implicitly given as an additional read-only state variable for concurrent components. The right-hand side of the figure shows the same counter after applying Lipton's theory of reductions. The increment operation consists of one atomic block that blocks until the mutex is free and afterwards the entire operation is performed in one indivisible step. Note that in this component the invariant that the mutex is free always holds, since the single step of the operation `inc` starts and ends in a state where the mutex is free. Therefore,

data type `mutex` = `free` | `locked`(*Tid*: `ThreadId`)

`mutex_lock`(*Tid*; *Mut*)
  atomic *Mut* = `free` { *Mut* := `locked`(*Tid*) }
`mutex_unlock`(*Tid*; *Mut*)
  assert *Mut* = `locked`(*Tid*); *Mut* := `free`

data type `rwlock` = `readers`(`tids`: `Set`⟨`ThreadId`⟩) | `writer`(*Tid*: `ThreadId`)

`rwlock_rlock`(*Tid*; *Rwl*)
  atomic *Rwl*.`readers`? { *Rwl* := `readers`(*Rwl*.`readers` ++ *Tid*) }
`rwlock_wlock`(*Tid*; *Rwl*)
  atomic *Rwl* = `readers`(∅) { *Rwl* := `writer`(*Tid*) }
`rwlock_runlock`(*Tid*; *Rwl*)
  atomic *Rwl*.`readers`? ∧ *Tid* ∈ *Rwl*.`readers` { *Rwl* := `readers`(*Rwl*.`readers` -- *Tid*) }
`rwlock_wunlock`(*Tid*; *Rwl*)
  assert *Rwl* = `writer`(*Tid*);*Rwl* := `readers`(∅)

Figure 13.2: Mutex and Reader/Writer-Lock Specification with Atomic Blocks

the implementation can be simplified further by removing the guard of the atomic block. The component *Counter*′ is a specification component in the sense that every operation takes exactly one atomic step. Thus, sequential reasoning is applicable to show for example that *Counter*′ refines another component with a data refinement or crash refinement of Ch. 5.

Lipton [87] developed conditions, which are sufficient to combine several statements into one atomic block. The idea is that statements of one thread may be reordered in the global interleaving of the component, if they *commute* with all other statements of all other threads. Lipton distinguished statements that commute to the left, right and in both directions, and called them *left-*, *right-* and *both-mover* accordingly. Acquiring a mutex or reader/writer lock is a right-mover and releasing is a left-mover. If locks are used correctly then all statements in the critical section are both-movers. If in a program $\{p; q\}$ the statement p is a right-mover, then it is always possible to construct an execution of all threads with the property that the transitions of p are directly adjacent to the first transition of q.

Fig. 13.3 shows an example with two threads. The first executes $\{p; q\}$ and is depicted in **red** in the figure. The other thread executes $\{p'; q'\}$. The figure shows an interleaving $I = (s_0, s_1, s_2, s_3, s_4)$ of the entire system at the bottom. The first thread is scheduled for the first and last transition. If p is a right-mover, then the alternative interleaving $I' = (s_0, s'_1, s'_2, s_3, s_4)$ shown in the figure also exists. Note that the alternative scheduling reaches the same final state, and each of the statements p', q' and p has the same effect (due to commutation), just in a different order. Thus, the observable behavior of both interleavings is the same. Interestingly, the alternative interleaving, or more precisely the interval $I'' = (s_0, s'_1, s'_2, s_4)$, is an interleaving of another system, namely the system where the first thread executes the program **atomic** $\{p; q\}$. Note that this contraction of two transitions into one also preserves observable behavior. The knowledge that a statement is a mover in a certain direction allows us to systematically reduce the set of interleavings that need to be considered during verification.

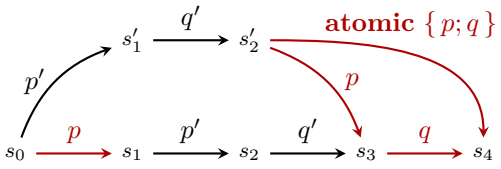

Figure 13.3: Program p commutes to the right of p' and q'

In order to prove that a program p commutes to the right, the proof obligation (right mover) must be shown for all statements p' of other threads where $\underline{x}$ are the free variables of p and

concurrent component *Counter*
state $Cnt\colon \mathbb{N}, Mut\colon$ `mutex`

interface operations
`inc`(; Cnt_0)

`mutex_lock`(Tid; Mut);
$Cnt_0 := Cnt$;
Cnt_0 :+= 1;
$Cnt := Cnt_0$;
`mutex_unlock`(Tid; Mut);

concurrent component *Counter′*
state $Cnt\colon \mathbb{N}, Mut\colon$ `mutex`
invariant $Mut =$ `free`
interface operations
`inc`(; Cnt_0)
atomic $Mut =$ `free` {
`mutex_lock`(Tid; Mut);
$Cnt_0 := Cnt$;
Cnt_0 :+= 1;
$Cnt := Cnt_0$;
`mutex_unlock`(Tid; Mut);
}

Figure 13.4: A Concurrent Counter before (left) and after (right) Applying Lipton Reductions

p'.

$$\langle p; p'\rangle\ \underline{x} = \underline{x}_1 \vdash \langle p'; p\rangle\ \underline{x} = \underline{x}_1 \qquad \text{(right mover)}$$

The proof obligation omits several details discussed in Sec. 13.3.

For lock-based programs oftentimes it is not necessary to prove that the alternative interleaving I' and I'' exists for the initial interleaving I, but that the initial interleaving does not exist in the first place and we can therefore safely assume that parts of the program are executed atomically. More formally, we will usually show that the antecedent of proof obligation (right mover) already evaluates to *false*. The basic idea is to use assertions φ and ψ that hold before the execution of p and p', respectively, and to show that if assertion φ holds in an initial state s and p is executed, then the resulting state can not satisfy the assertion ψ, too.

`inc`(; Cnt_0)
`mutex_lock`(Tid; Mut);
assert $Mut =$ `locked`(Tid); $Cnt_0 := Cnt$;
assert $Mut =$ `locked`(Tid); Cnt_0 :+= 1;
assert $Mut =$ `locked`(Tid); $Cnt := Cnt_0$;
`mutex_unlock`(Tid; Mut);

Figure 13.5: Assertions for the Component *Counter*

Now reconsider the example of a concurrent counter in Fig. 13.4. The component annotated with assertions is shown in Fig. 13.5. We first show with the invariant proofs of Ch. 4 that the assertions hold. Afterwards, we prove that all statements in the critical section are both-movers.

More formally, we prove that each of the programs[1]

$$\begin{aligned} p_0 &\equiv \{\,\textbf{assert}\ mut = \texttt{locked}(tid_0);\ cnt_0 := cnt;\,\},\\ p_1 &\equiv \{\,\textbf{assert}\ mut = \texttt{locked}(tid_0);\ cnt_0 :+= 1;\,\}, \quad \text{and}\\ p_2 &\equiv \{\,\textbf{assert}\ mut = \texttt{locked}(tid_0);\ cnt := cnt_0;\,\} \end{aligned}$$

executed by the first thread commutes in both directions with every program

$$\begin{aligned} p'_0 &\equiv \{\,\textbf{assert}\ mut = \texttt{locked}(tid_1);\ cnt'_0 := cnt;\,\},\\ p'_1 &\equiv \{\,\textbf{assert}\ mut = \texttt{locked}(tid_1);\ cnt'_0 :+= 1;\,\}, \quad \text{and}\\ p'_2 &\equiv \{\,\textbf{assert}\ mut = \texttt{locked}(tid_1);\ cnt := cnt'_0;\,\} \end{aligned}$$

executed by some other thread. Note that the thread identifier tid_1 used in the programs p'_0, p'_1 and p'_2 is distinct from the thread identifier tid_0, and that local variables, such as cnt_0,

[1] Note that lower case, static variables are used, because dynamic logic only proof obligations only use static variables

are renamed properly. With these programs proof obligation (right mover) is trivial, since the antecedent is inconsistent. It is not possible to satisfy the formula

$$mut = \texttt{locked}(tid_0) \wedge mut = \texttt{locked}(tid_1)$$

given $tid_0 \neq tid_1$. Note that p_0, p_1 and p_2 trivially commute over the calls to `mutex_lock` and `mutex_unlock`, because the assigned variables are disjoint.

After these proofs, we annotate statements and programs p with the their mover type, i.e., we write $p\colon \mathbb{L}$, $p\colon \mathbb{R}$ and $p\colon \mathbb{B}$ if we have proven that p is a left-, right- and both-mover, respectively. If we do not yet know whether p commutes at all, but it is a statement executed atomically, we write $p\colon \mathbb{A}$. Fig. 13.6 shows this for the example of the concurrent counter. Lipton observed that a sequence $\{\, p\colon \mathbb{R}; q\colon \mathbb{R}; q'\colon \mathbb{L} \,\}$ can be replaced by an atomic section with the blocking condition of the atomic section p, if the statements q and q' do not block. This step yields the component *Counter*′ shown in Fig. 13.4 on page 174, i.e., we have reached a specification component for the example.

```
inc(; Cnt0)
  mutex_lock(Tid; Mut) : ℝ;
  Cnt0 := Cnt : 𝔹;
  Cnt0 :+= 1 : 𝔹;
  Cnt := Cnt0 : 𝔹;
  mutex_unlock(Tid; Mut) : 𝕃;
```

Figure 13.6: Mover Annotations for the Component *Counter*

13.3 Atomicity Refinement with Lipton Reductions

The approach of Lipton [87] facilitates composing blocks that are atomic with respect to concurrency, i.e., it shows that the input/output behavior of a concurrent component does not change when several successive statements executed by one thread are replaced by an atomic block consisting of those statements. However, it does not show that the behavior during a power failure is preserved. In this section we integrate Lipton reductions in the calculus for atomicity refinement of Ch. 5, in order to increase the atomicity of concurrent components, until we have reached a specification component or are at least close to one.

First, let us revisit the argument of commuting program steps of different threads in the context of power failures and a non-retracting component (see Def. 4.4 on page 28). Fig. 13.7 shows the initial execution $I = (s_0, \ldots, s_i, s_{i+1}, \ldots, s_n, s_{n+1})$ at the bottom. The first thread executes the program $\{\, p; q \,\}$ and is again depicted in **red**. Other threads execute the statements p'. Assume that we have already proven that p is a right-mover and that q is a left-mover. We are now trying to find an argument why we may replace the program $\{\, p; q \,\}$ by **atomic** $\{\, p; q \,\}$. Since p is a right-mover, a state s'_{n-1} exists with the property that it is reached from s_i by the statements q' of the other threads and afterwards the program p again yields state s_n. The question now is why it is permissible to either remove p from the execution before the reset transition or add q to the execution, i.e., why the dashed arrows in the figure exist. Otherwise, we have not constructed the necessary execution of **atomic** $\{\, p; q \,\}$ and are therefore not allowed to replace $\{\, p; q \,\}$ with an atomic block. However, we have already seen criteria that ensure the existence of alternative executions in Ch. 5. If p is crash-recover-retractable (see Def. 5.9 on page 54), then the transition $s'_{n-1} \xrightarrow{\text{Reset}} s_{n+1}$ exists. If on the other hand q is crash-recover-introducible (see Def. 5.10 on page 54), then the transitions $s_n \xrightarrow{q} s'_n \xrightarrow{\text{Reset}} s_{n+1}$ exists.

In the following, we will formalize this insight and derive a calculus with Lipton reductions for concurrent, crash-aware components. First, we need to define the set of programs p and q where commutation is possible.[2]

[2]Lower case, static variables are used in Def. 13.1, because the programs are used in dynamic logic proof obligations.

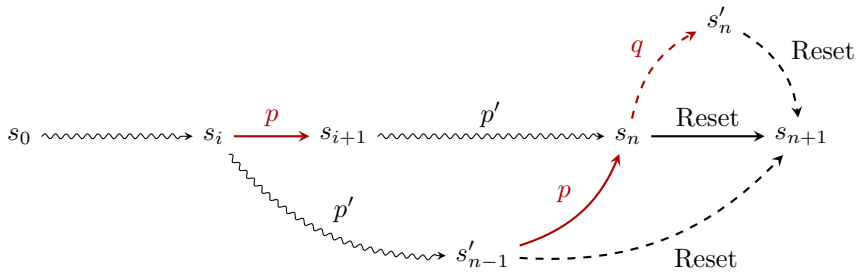

Figure 13.7: Lipton Reductions & Crashes

Definition 13.1 (Atoms of Programs & Components)**.** The set of atoms of a regular, sequential program p, written $At(p)$, contains a program with its associated assertion for every non-stuttering transition of the program p, i.e., for every $I \models p$ and every $i < \#\, I$ there is a program q with $I_{[i..i+1]} \models q$ and $q \in At(p)$ for every non-stuttering transition $I_{[i..i+1]}$.[3] The set of atoms is defined by the equations

$$
\begin{aligned}
&At(\textbf{assert}\ \varphi; \underline{x} \coloneqq \underline{t}) = \{\ \textbf{assert}\ \varphi; \underline{x} \coloneqq \underline{t}\ \}\\
&At(\textbf{assert}\ \varphi; \textbf{if}\ \psi\ \textbf{then}\ p\ \textbf{else}\ q) = \{\ \textbf{assert}\ \varphi; b \coloneqq \psi\ \} \cup At(p) \cup At(q)\\
&At(\textbf{assert}\ \varphi; \textbf{while}\ \psi\ \textbf{do}\ \{\, p; \textbf{assert}\ \varphi'\,\}) = \{\ \textbf{assert}\ \varphi \vee \varphi'; b \coloneqq \psi\ \} \cup At(p)\\
&At(\textbf{assert}\ \varphi; \texttt{Proc}(\underline{t}; \underline{x})) = \{\ \textbf{assert}\ \varphi; y' \coloneqq \underline{t}\ \} \cup At(p\{\underline{z} \mapsto \underline{x}\})\\
&At(\textbf{assert}\ \varphi; \textbf{atomic}\ \psi\ \{\, p\,\}) = \{\ \textbf{assert}\ \varphi; \textbf{atomic}\ \psi\ \{\, p\,\}\}\\
&At(\textbf{assert}\ \varphi; \textbf{choose}\ \underline{x}\ \textbf{with}\ \varphi\ \textbf{in}\ p\ \textbf{ifnone}\ q) = At(p\{\underline{x} \mapsto \underline{y}'\}) \cup At(q) \cup\\
&\quad \cup \{\ \textbf{choose*}\ \underline{y}'\ \textbf{with}\ \varphi^{\underline{y}'}_{\underline{x}}\ \textbf{in}\ \underline{y}'' \coloneqq \underline{y}', b \coloneqq \mathit{true}\ \textbf{ifnone}\ b \coloneqq \mathit{false}\}\\
&At(p; q) = At(p) \cup At(q)
\end{aligned}
$$

for non-recursive procedures `Proc` with declaration $\texttt{Proc}(\underline{y}; \underline{z})\ \{\, p\,\}$ and globally fresh variables b, b' and $\underline{y}'$. If the definition is not applicable to a program p we add an assertion of *true*, i.e., $At(p)$ is defined as $At(\textbf{assert}\ \mathit{true}; p)$ in this case.

The atoms of a component C, written $At(C)$ is the union of the sets of atoms for the program of every interface and internal operation.

Def. 13.1 makes tests and choices visible by assigning a globally fresh variable b accordingly. For the program construct **choose** the variables need to be renamed in order to avoid clashes and the choice of variables $\underline{y}'$ is now observable, because the (free) variables $\underline{y}''$ carry their value afterwards. Stuttering transitions, such as assertion failures or blocking, are omitted in the atoms of a program, because other programs trivially commute over these steps.

Next we define the calculus to determine whether a program p with $p \in At(C)$ is a left-, right-, both- or none-mover. Fig. 13.8 shows the rules of this calculus. The variables $\underline{x}$ are the state variables of the component C and all its subcomponents. In contrast, $\underline{y}$ denotes all free variables of p and q. The first premise asserts the criterion for crash atomicity. The second premise of each of the rules is a dynamic logic proof obligation which ensures that the program p commutes in the desired direction. The predicate R again denotes the crash and subsequent recovery of component C, see (*R*-is-*crash*-`recover`) on page 50.

We distinguish the two types of right-movers $\overset{\frown}{\mathbb{R}}$ and $\overset{\frown}{\mathbb{R}}$ as separate judgments, because they do not compose sequentially, i.e., in a program $\{\, p; q\,\}$ where $\mathit{inv} \vdash p : \overset{\frown}{\mathbb{R}}$ and $\mathit{inv} \vdash q : \overset{\frown}{\mathbb{R}}$ holds, the entire program is atomic with respect to concurrency, however not atomic with

[3] More precisely, the program q must satisfy $I_{[i..i+1]} \models \textbf{choose}\ \underline{x}\ \textbf{in}\ \{\, q\,\}$, where $\underline{x}$ are the variables, which are locally bound by **choose**-statements or procedure calls around p.

$$\frac{inv \vdash p : \curvearrowright_R \qquad \begin{array}{l}\text{for all } q \in At(C):\\ tid \neq tid', inv(\widehat{\underline{x}}), \langle q\{tid \mapsto tid'\}; p\rangle\ \underline{y}' = \underline{y}\\ \vdash \langle p; q\{tid \mapsto tid'\}\rangle\ \underline{y}' = \underline{y}\end{array}}{inv \vdash p : \mathbb{L}}\ \begin{array}{l}\mathbb{L}\text{-Mover},\\ p \in At(C)\end{array}$$

$$\frac{inv \vdash p : \curvearrowright_R \qquad \begin{array}{l}\text{for all } q \in At(C):\\ tid \neq tid', inv(\widehat{\underline{x}}), \langle p; q\{tid \mapsto tid'\}\rangle\ \underline{y}' = \underline{y}\\ \vdash \langle q\{tid \mapsto tid'\}; p\rangle\ \underline{y}' = \underline{y}\end{array}}{inv \vdash p : \overset{\frown}{\mathbb{R}}}\ \begin{array}{l}\overset{\frown}{\mathbb{R}}\text{-Mover},\\ p \in At(C)\end{array}$$

$$\frac{inv \vdash p : \curvearrowleft_R \qquad \begin{array}{l}\text{for all } q \in At(C):\\ tid \neq tid', inv(\widehat{\underline{x}}), \langle p; q\{tid \mapsto tid'\}\rangle\ \underline{y}' = \underline{y}\\ \vdash \langle q\{tid \mapsto tid'\}; p\rangle\ \underline{y}' = \underline{y}\end{array}}{inv \vdash p : \overset{\frown}{\mathbb{R}}}\ \begin{array}{l}\overset{\frown}{\mathbb{R}}\text{-Mover},\\ p \in At(C)\end{array}$$

$$\frac{inv \vdash p : \mathbb{L} \quad inv \vdash p : \overset{\frown}{\mathbb{R}}}{inv \vdash p : \mathbb{B}}\ \mathbb{B}\text{-Mover} \qquad \frac{\varphi, inv \vdash \underline{x} \ := \ \underline{t} : \curvearrowright_R}{inv \vdash \{\,\textbf{assert}\ \varphi; \underline{x} \ := \ \underline{t}\,\} : \mathbb{A}}\ \mathbb{A}\text{-Assign}$$

$$\frac{\varphi, inv \vdash \textbf{atomic}\ \{\,p\,\} : \curvearrowright_R}{inv \vdash \{\,\textbf{assert}\ \varphi; \textbf{atomic}\ \{\,p\,\}\,\} : \mathbb{A}}\ \begin{array}{l}\mathbb{A}\text{-Atomic, if component } C\\ \text{of } p \text{ divergence-free}\end{array}$$

$$\frac{\text{for all } q \in At(p): \quad inv \vdash q : \dagger}{inv \vdash p : \dagger}\ \begin{array}{c}\dagger\text{-Mover, for } p \notin At(C),\\ p \text{ regular and sequential,}\\ \dagger \in \{\mathbb{L}, \overset{\frown}{\mathbb{R}}, \overset{\frown}{\mathbb{R}}, \mathbb{B}\}\end{array}$$

Figure 13.8: Calculus for Left/Right/Both/None-Movers

respect to crashes, since we may neither retract p nor introduce q in an execution where only p is executed before the crash.

The rules for atomic none-movers ($\mathbb{A}$) just show that a program is executed in one atomic step, which is the case if it is an assignment or an atomic statement without guard in a component C that is proven to be divergence-free, i.e., does not have any infinite computations. For programs that are not executed in one atomic step, the rule †-*Mover* lifts the judgment to regular, sequential programs p.

As we have already seen in Ch. 6, we prove that all steps of a component are crash-recover-introducible, which immediately proves the first premise for any judgment $inv \vdash p : \mathbb{L}$, $inv \vdash p : \overset{\frown}{\mathbb{R}}$ and $inv \vdash p : \mathbb{A}$.

Def. 13.2 defines the sequences of statements that may be combined into an atomic block.

Definition 13.2 (Reducible Program)**.** A program p is *reducible* if every finite sequence of atoms $p_0; \ldots; p_n$ it executes can be split into a (possibly empty) prefix $p_0, \ldots, p_i$ and a (possibly empty) suffix $p_{i+1}, \ldots p_n$ that satisfy (1.) to (4.).

1. $inv \vdash \{\,p_1; \cdots; p_i\,\} : \overset{\frown}{\mathbb{R}}$ or $inv \vdash \{\,p_1; \cdots; p_i\,\} : \overset{\frown}{\mathbb{R}}$ holds, and
2. $inv \vdash p_{i+1} : \mathbb{A}$ or $inv \vdash p_{i+1} : \mathbb{L}$ holds, and

3. $inv \vdash \{ p_{i+2}; \cdots; q_n \} : \mathbb{L}$ holds, and
4. the atoms p_i for $2 \leq i \leq n$ never block

Several examples should illustrate the concept of a reducible program. The program

$$p \equiv p_1; p_2 \equiv \textbf{atomic}\ \varphi\ \{ x\ :=\ 1 \} : \overset{\frown}{\mathbb{R}}; x\ :=\ 2 : \mathbb{L}$$

is reducible, because the only sequence of atoms is given by $p_1; p_2$ itself, the sequence obviously has a corresponding split into a prefix and suffix p_2 never blocks. More interesting is the case of an if-statement where one branch is a none-mover. For example the program

$$\begin{array}{l}
\textbf{atomic}\ \varphi\ \{ x\ :=\ 1 \} : \overset{\frown}{\mathbb{R}}; \\
\textbf{if}\ \psi\ \textbf{then} \\
\quad y\ :=\ 2 : \mathbb{A}; \\
\textbf{else} \\
\quad x\ :=\ 3 : \mathbb{L}; \\
\quad x\ :=\ 4 : \mathbb{L};
\end{array}$$

is also reducible assuming $b\ :=\ \varphi : \overset{\frown}{\mathbb{R}}$ holds. The program has the sequence

$$\textbf{atomic}\ \varphi\ \{ x\ :=\ 1 \} : \overset{\frown}{\mathbb{R}}; b\ :=\ \varphi : \overset{\frown}{\mathbb{R}}; y\ :=\ 2 : \mathbb{A}$$

and the sequence

$$\textbf{atomic}\ \varphi\ \{ x\ :=\ 1 \} : \overset{\frown}{\mathbb{R}}; b\ :=\ \varphi : \overset{\frown}{\mathbb{R}}; x\ :=\ 3 : \mathbb{L}; x\ :=\ 4 : \mathbb{L};$$

as the only possible sequences of atoms. Each sequence admits a split that satisfies Def. 13.2.

Thm. 16 extends the atomicity refinement of Ch. 5 to non-retracting, concurrent components.

Theorem 16 (Atomicity Refinement of Retraction-Free, Concurrent Components)**.** *Given a reducible program p of a non-retracting and divergence-free component C with concurrent invariant inv, then*

$$C\{p \mapsto \textbf{\textit{atomic}}\ \varphi\ \{ p \}\} \sqsubseteq C$$

holds, where φ is the guard of the first statement of p if it is an atomic block or $\varphi \equiv true$ if the first statement is not an atomic block, and $C\{p \mapsto \textbf{\textit{atomic}}\ \varphi\ \{ p \}\}$ is divergence-free with invariant inv.

Proof. Divergence-freedom of C ensures that we may assume that every execution of p reachable in C terminates.

Given a finite execution I of the concurrent component C. The proof considers two cases.

First, assume $inv \vdash \{ p_1; \cdots; p_i \} : \overset{\frown}{\mathbb{R}}$ holds where i is chosen according to Def. 13.2. For all partial executions of p we add a crash-recover-subsuming execution, which is guaranteed by the definition of crash-recover-introducibility (Def. 5.10 on page 54), for the remaining steps of p, either at the end of I if no reset transition follows or before the next reset transitions, and get an interval I'. By the choice of the additional steps, the reset transitions remain valid. We then commute the programs in the corresponding directions towards p_{i+1} until they are executed in direct sequence. A contraction of these transitions into one yields the final interval I'' that is a run of the component $C\{p \mapsto \textbf{atomic}\ \varphi\ \{ p \}\}$.

Now consider the case where $inv \vdash \{ p_1; \cdots; p_i \} : \overset{\frown}{\mathbb{R}}$ holds. Here we commute all partial executions where p_{i+1} was never reached towards the next reset transition (or at the end

of the interval). We then remove these steps from the interval and get an interval I'. The reset transitions still remain valid, because all p_j with $0 \leq j \leq i$ are crash-recover-retractable. Afterwards, we continue as in the first case.

For infinite executions I we may assume that there are no partial executions of p after the last reset transition, because we assume weak-fair scheduling as the semantics of concurrent components. Partial executions before the last reset are treated similarly as in the finite case. □

Note that Thm. 16 does not yet cover retracting components. However, it is possible to extend the proof obligations in future work as explained in the following. Two issues need to be addressed. First, commutations may not introduce additional synchronized states into the run. Take Fig. 13.3 on page 173 where program p is a right-mover as an example. We need to ensure that state s'_1 is only synchronized if s_2 is and that state s'_2 is only synchronized if state s_3 is. Then all retraction possible in the original run in the figure are also possible in the alternative run, with an additional retraction of the operation p. Basically, right-movers may be re-ordered to the right of synchronized states. For left-movers it needs to be shown that they are never re-ordered to the left of a synchronized state. Second, if additional transitions are added to the run, then they may not add synchronized states, i.e., the definition crash-recover-introducible needs to be amended by also showing that the crash-recover-subsuming execution only "weakens" synchronization, and therefore that in the example of Fig. 13.7 on page 176 the additional transition of q does not introduce an additional synchronized state s'_n, if s_n is not synchronized. This extension is, however, not in the scope of this thesis.

The theorem ensures that the innermost lock *mut* can be replaced by an atomic section with a guard $mut = \texttt{free}$. The resulting component usually has the invariant $mut = \texttt{free}$, if locks are nested properly. With Lem. 13.3 we can remove this guard entirely without altering the semantics.

Lemma 13.3 (Removal of Atomic Guards)**.** *If component C has a concurrent invariant inv with $inv(\widehat{\underline{x}}) \vdash \varphi$, then $C\{\textbf{\textit{atomic}}\ \varphi\ \{\,p\,\} \mapsto \textbf{\textit{atomic}}\ \{\,p\,\}\} \sqsubseteq C$ holds.* □

Afterwards, the atomic section **atomic** $\{\,p\,\}$ is an atomic none-mover ($\mathbb{A}$) and further applications of Thm. 16 are possible with q' of the theorem chosen as $q' \equiv$ **atomic** $\{\,p\,\}$.

13.4 Ownership Annotations & Invariant Expressions

In practice, adding assertions to a large component and proving commutations is rather cumbersome. We automate this process by adding an ownership annotation to the entire component. In the example of the concurrent counter Sec. 13.2 the annotation (ownership-counter) is added.

$$\mathit{Cnt} \quad \textbf{owned by} \quad \mathit{Mut}.\texttt{ownership} \tag{ownership-counter}$$

The general form of the annotation is shown in (ownership), where the free variables of the left-hand side a are the state variables $\underline{X}$ of the component and some additional variables $\underline{y}$.

$$a(\underline{X}, \underline{y}) \quad \textbf{owned by} \quad o(\underline{X}, \underline{y}) \tag{ownership}$$

The expression $a(\underline{X}, \underline{y})$ must be a readable and writable location, i.e., it must be allowed as a left-hand side of an assignment. Equation (accessor-form) gives a grammar for the allowed expressions. Essentially, a variable x, access to a field $a.\texttt{field}$ of an instance a of a data type, access to the element $a[n]$ of an array a, and access to the value $a[key]$ of a map a under the key key is allowed.

$$a \coloneqq \quad x \quad | \quad a.\texttt{field} \quad | \quad a[n] \quad | \quad a[key] \tag{accessor-form}$$

Another example for a potential left-hand side is the access to the mapping of the erase block manager of Ch. 10. There $Vols[v][l]$ is used to access the physical erase block number of the logical erase block $\langle v, l \rangle$.

The right-hand side $o(\underline{X}, \underline{y})$ of the ownership annotation (ownership) may only use the state variables $\underline{X}$ and the variables $\underline{y}$ bound by the left-hand side of the ownership annotation. Furthermore, the type of expression $o(\underline{X}, \underline{y})$ must be the type `Owner`.

We call the right-hand side the *owner expression* of the field on the left-hand side. The idea is that the owner expression stores, which thread currently has access privileges for the field. A thread with identifier may only read or write the field if the owner expression of the field grants access.

data type `Owner` = `readers(tids: Set⟨ThreadId⟩)` | `writer(tid: ThreadId)`

Owner expressions allow only a single writer and multiple readers concurrently, by definition. We write `read`?(Tid, o) and `write`?(Tid, o), if the owner expression o grants thread Tid read and write access, respectively.

$$\texttt{read}?(Tid, o) \leftrightarrow \begin{cases} Tid \in tids & \text{if } o = \texttt{readers}(tids) \\ Tid = Tid' & \text{if } o = \texttt{writer}(Tid') \end{cases}$$

$$\texttt{write}?(Tid, o) \leftrightarrow o = \texttt{writer}(Tid)$$

Mutex and reader/writer locks are the standard way to acquire and release ownership. We therefore define a canonical way to derive an owner expression from a mutex mut and reader/writer lock rwl with the function `ownership`.

$$mut.\texttt{ownership} = \begin{cases} \texttt{readers}(\emptyset) & \text{if } mut = \texttt{free} \\ \texttt{writer}(Tid) & \text{if } mut = \texttt{locked}(Tid) \end{cases}$$

$$rwl.\texttt{ownership} = \begin{cases} \texttt{readers}(tids) & \text{if } rwl = \texttt{readers}(tids) \\ \texttt{writer}(Tid) & \text{if } rwl = \texttt{writer}(Tid) \end{cases}$$

Consider the example of the concurrent counter of Fig. 13.4 on page 174 and its ownership annotation (ownership-counter) from above. The ownership annotation states that if the mutex is locked, then the thread holding the mutex has acquired read and write ownership for the state variable cnt. Conversely, once a thread releases the mutex, the thread relinquishes its ownership, too.

The annotations are leveraged in three ways. First, they allow for canonical assertions that ensure that for each access of a field of a thread, the thread has the required ownership. Second, two canonical rely conditions are implied by the annotations, namely if a thread has read access for a field, then the field is unchanged, and that other threads Tid' may not change the ownership the thread Tid has. Third, if a program p only accesses fields with an owner expression (and does not change the value of the owner expression), local variables or the input and output variables to an operation, then the program is immediately a both mover, i.e., $inv \vdash p : \mathbb{B}$ holds (assuming that p is also introducible). Another advantage of ownership annotations is that they imply that the program is *data-race free*, i.e., that there are no two concurrent writers to a field, which can not directly be proven directly with the rely/guarantee calculus, since the calculus can not detect assignments of the form $\underline{x} := \underline{x}$.

Ownership Assertions We first discuss briefly the process of adding ownership assertions based on an ownership annotation of the form (ownership) for regular, sequential programs. For each read access $a(\underline{X}, \underline{t})$, where $\underline{t}$ is a substitution for $\underline{y}$ of the ownership annotation (ownership) in conditions and guards of **if**-, **while**- and **choose** statements, an assertion of the form

assert `read`?$(Tid, o(\underline{X}, \underline{t}))$

is added in front of the statement itself and as the first statement in its body, i.e., for **if**-statements the assertion is added in the if-branch and in the else-branch. For **while**-loops the assertion is also added at the end of the body. The guard of atomic statements may not contain fields with an owner expression. And **choose**-statements may not quantify over fields with an owner expression. For example the statement

$$\textbf{choose}\ v, l\ \textbf{with}\ \mathit{Vols}[v][l] = \ldots\ \textbf{in}\ \{\, p \,\}$$

is disallowed, if there is an owner expression for $\mathit{Vols}[v][l]$, since it is not possible to find out for which fields $\mathit{Vols}[v][l]$ access is required. For assignments $a(\underline{X}, \underline{t}) := t'$ where the left-hand side has an owner expression, the assertion

$$\textbf{assert}\ \texttt{write}?(\mathit{Tid}, o(\underline{X}, \underline{t}))$$

is added before and after the assignment. Additional assertions are added for read accesses in $a(\underline{X}, \underline{t})$ and t'. For calls $\texttt{Proc}(\underline{t}; \underline{X}, \underline{z})$ where $\underline{X}$ are the state variables of the component C and the call is within the component C, assertions for read accesses in $\underline{t}$ are added. Note that $\underline{z}$ can not contain a field with an ownership expression, since it can not contain any state variables of the component. For calls $\texttt{Proc}(\underline{t}; \underline{Y}, \underline{z})$ to a subcomponent A with state variables $\underline{Y}$, the output parameters $\underline{z}$ may contain state variables of C with an owner expression and an assertion for write access is added, additionally a proof obligation is generated, which ensures that the write access is maintained throughout the entire call.

Assertions are also added to the body of atomic sections and atomic sections with a body that modifies an ownership expression is disallowed. Note that the restrictions introduced here are usually given in implementation components, and therefore do not limit the approach.

Rely Condition The canonical rely condition associated with an ownership annotation (ownership) is given by formula (ownership-rely). The formula states that if the thread has read access for a field, then the field is unchanged over the steps of other threads. Similarly, other threads do neither add nor remove read or write access privileges from a thread.

$$\begin{aligned} &\big(\forall \underline{y}.\ \texttt{read}?(o(\underline{X}', \underline{y}), \mathit{Tid}') \to a(\underline{X}', \underline{y}) = a(\underline{X}'', \underline{y})\big) \\ \wedge\ &\big(\forall \underline{y}.\ \texttt{read}?(o(\underline{X}', \underline{y}), \mathit{Tid}') \leftrightarrow \texttt{read}?(o(\underline{X}'', \underline{y}), \mathit{Tid}')\big) \\ \wedge\ &\big(\forall \underline{y}.\ \texttt{write}?(o(\underline{X}', \underline{y}), \mathit{Tid}') \leftrightarrow \texttt{write}?(o(\underline{X}'', \underline{y}), \mathit{Tid}')\big) \end{aligned} \qquad \text{(ownership-rely)}$$

Both-Mover Once the invariant proofs of Ch. 4 are completed and the above assertions hold, all programs p that only access fields with an owner expression, local variables or the input/output parameters of the operation are both-movers, i.e., $\mathit{inv} \vdash p : \mathbb{B}$ holds, assuming that $\varphi, \mathit{inv} \vdash p : \curvearrowright$ is already proven.[4] The reason is that the assertions prove that before and after every step each thread has the necessary permissions to access the fields. If steps of separate threads are adjacent in the global interleaving then they must both be readers of all the fields. Then the two steps obviously commute.

In the example of the counter of Fig. 13.5 on page 174, it only needs to be shown explicitly that the judgments

$$\texttt{mutex\_lock}(\mathit{Tid}; \mathit{mut}) : \mathbb{R} \qquad \text{and} \qquad \texttt{mutex\_unlock}(\mathit{Tid}; \mathit{mut}) : \mathbb{L}$$

hold. The final component $\mathit{Counter}'$ shown in Fig. 13.4 on page 174 is then inferred automatically.

[4] Note that allocation and deallocation of fields is omitted from the discussion and these operations are not automatically both-movers. In concurrent components these operations need to be marked as such. Then only ownership directly after the allocation and directly before the deallocation is asserted, and it is not automatically inferred that the statement is a both-mover.

13.5 Related Work

Refinement and abstraction of atomicity is quite common for concurrent systems. Thm. 16 is essentially an extension of Lipton reductions [87] to crash-aware, concurrent systems. The refinement calculus of Back [14] uses the opposite direction. It starts out with an atomic program and splits it into smaller actions in refinement steps.

The calculus of atomic actions due to Elmas et al. [42] is also an extension of Lipton's approach for highly concurrent, linearizable programs. The calculus does not consider crashes. However, it provides a more incremental verification methodology than the calculus of this chapter for highly concurrent systems. The assertions and invariants are incrementally validated in [42], whereas here a rely/guarantee proof is used to validate them before applying any reductions. The reason is that for the lock-based concurrency considered in this thesis, a more automated approach that automatically detects both-movers based on ownership annotations, is preferable to the fine-grained approach of the calculus of atomic actions. The latter is more geared towards highly concurrent algorithms and data structures. Several rules of the calculus in [42] do not preserve liveness. However, liveness is critical for the existence of the crash-recover-subsuming execution used in Ch. 5 and Ch. 13 to increase the atomicity with respect to power failures. A combination of the incremental approach of the calculus of atomic actions and the additional criteria for reductions as discussed in this thesis is an interesting avenue for future research. This is especially relevant with respect to the emerging technology of byte-addressable Non-volatile Random Access Memory (= NVRAM). NVRAM allows algorithms with the high degree of concurrency of traditional linearizable and lock-free data structures to operate on persistent memory.

In the context of algorithms and data structures for NVRAM, in [69] the correctness criterion (buffered) durable linearizability is introduced. It is proven that any non-blocking data-race free program can be transformed into a program that is buffered durably linearizable. A proof method of persist points, similar to linearization points, is proposed and proven correct in [69], too. The method proposed in this thesis to increase the atomicity is orthogonal to a direct proof of (buffered) durable linearizability with persist points.

Rely/Guarantee reasoning has the advantage that only linearly many proof obligations need to be shown whereas for a direct proof of commutation in the calculus of atomic actions or the Owicki-Gries-Method [104, 105] a proof of quadratically many proof obligations is necessary.

Ownership annotations are used in the C verifier VCC [32] and Spec# [70] in order to ensure data-race freedom of the code. Fractional permissions [19] in concurrent versions of separation logics [114] serve a similar purpose and are for example supported by the C code verifier VeriFast [71].

14 Concurrent Wear-Leveling

Summary. This Chapter applies the specification and proof methodology of Ch. 13 to the model of the erase block manager of Ch. 10. Ownership annotations and additional rely conditions for the specification components *AEBM* and *EBM Headers* are discussed. The implementation of the erase block manager is augmented with locks to guarantee correct synchronization between threads. It is proven that atomicity refinement of Ch. 13 with Lipton reductions yields a component that is almost atomic and where a standard forward simulation is applicable to prove refinement. In summary, this allows the Flashix file system to perform wear-leveling and asynchronous erasure in a separate thread, concurrently to read and write requests.

Contents

14.1 Specification of a Concurrent Erase Block Manager

The specification of a concurrent erase block manager is given by the component *AEBM* augmented with ownership annotations. Fig. 14.1 depicts the state and invariants of the component.

As an additional *ghost* state variable *ovols* is added, which stores the ownership information for each erase block by (ownership-*AEBM*), i.e., which thread currently has read or write access privileges.

$$avols[v][l] \quad \textbf{owned by} \quad ovols[v][l] \qquad \text{(ownership-}AEBM\text{)}$$

Ghost state is omitted during code generation and only used for the verification. The only restriction for ghost state is that it may not influence the control flow of the component and the data stored in the "normal" state. In the case of component hierarchies ghost state is used to propagate synchronization performed by a client component to the subcomponent. For example, a client of the erase block manager has to ensure that no two writes are performed on the same erase block concurrently. The reason is the limitation to sequential writes only. More formally, it is necessary to ensure that preconditions are stable over steps of other threads. Write access has the precondition that the offset of the write is above the field

concurrent component *AEBM*
state $avols\colon \mathbb{V} \nrightarrow \texttt{Array}\langle\texttt{Leb}\rangle$
ghost state $ovols\colon \mathbb{V} \nrightarrow \texttt{Array}\langle\texttt{Owner}\rangle$
ownership
$avols[v][l]$ **owned by** $ovols[v][l]$
concurrent invariant
$\texttt{avols-inv}(avols) \wedge dom(ovols) = dom(avols) \wedge \forall v.\ \#\, avols[v] = \#\, ovols[v]$
rely
$dom(avols'') = dom(avols') \wedge \forall v.\ \#\, avols''[v] = \#\, avols'[v]$
interface operations
$\texttt{aebm\_rlock}(v, l)$
pre $\langle v, l\rangle \in ovols$
if $ovols[v][l].\texttt{readers}?$ **then** $ovols[v][l] := \texttt{readers}(ovols[v][l].\texttt{tids} +\!\!+ tid)$;
$\texttt{aebm\_wlock}(v, l)$
pre $\langle v, l\rangle \in ovols$
if $ovols[v][l] = \texttt{readers}(\emptyset)$ **then** $ovols[v][l] := \texttt{writer}(tid)$;
$\texttt{aebm\_unlock}(v, l)$
pre $\langle v, l\rangle \in ovols$
if $tid \in ovols[v][l].\texttt{readers}$ **then** $ovols[v][l] := \texttt{unlock}(ovols[v][l], tid)$;
$\texttt{aebm\_read}(v, l, poff, boff, len; buf, err)$
pre $\langle v, l\rangle \in avols \wedge poff + len \leq \texttt{LEB\_SIZE} \wedge boff + len \leq \#\, buf \wedge \texttt{read}?(ovols[v][l], tid)$
$\{\ buf := \texttt{copy}(avols[v][l].\texttt{data}, poff, buf, boff, len),\ err := \texttt{ESUCCESS}\ \}$
$\vee\ \{\ \texttt{fail}(; err)\ \}$
$\texttt{aebm\_write}(v, l, poff, boff, len, buf; err)$
pre $\langle v, l\rangle \in avols \wedge avols[v][l].\texttt{mapped}? \wedge poff + len \leq \texttt{LEB\_SIZE} \wedge boff + len \leq \#\, buf$
$\wedge\ \texttt{page-aligned}(poff) \wedge \texttt{page-aligned}(len) \wedge avols[v][l].\texttt{written} \leq poff$
$\wedge\ \texttt{write}?(ovols[v][l], tid)$
$\{\ avols[v][l] := \texttt{mapped}(avols[v][l].\texttt{data}[0..poff] + buf[boff..(boff + len)]$
$+ \texttt{empty-array}(\texttt{LEB\_SIZE} - (poff + len)), poff + len);$
$err := \texttt{ESUCCESS}\ \}$
$\vee\ \{$ **choose** len_0 **with** $len_0 = 0 \vee len_0 < len \wedge \texttt{page-aligned}(len_0)$ **in**
if $len_0 \neq 0$ **then**
$avols[v][l] := \texttt{mapped}(avols[v][l].\texttt{data}[0..poff] + buf[boff..(boff + len_0)]$
$+ \texttt{empty-array}(\texttt{LEB\_SIZE} - (poff + len_0)), poff + len_0);$
$\texttt{fail}(; err)\ \}$
$\vdots$

Figure 14.1: Abstract Specification of a Concurrent Erase Block Manager

`written` of the logical erase block. This precondition is stable only if other write accesses are not issued by the client concurrently.

In order to acquire access to a logical erase block the additional operations `aebm_rlock`, `aebm_wlock`, `aebm_unlock` are added to the interface operations. Note that ownership is only acquired by a thread calling `aebm_rlock` and `aebm_wlock` if, in the context of the client, it can be shown that the condition $ovols[v][l].\texttt{readers}?$ and $ovols[v][l] = \texttt{readers}(\emptyset)$ of the **if**-statement hold at the time of the call, respectively. A similar case is discussed for the erase block manager in Sec. 14.3. The algebraic function $\texttt{unlock}(o, tid)$ removes the thread identifier tid as a reader or writer of the ownership field o.

The operations to read, write, map, unmap and erase a logical erase block then have as additional precondition that the calling thread has the required ownership. For example for the operation `aebm_read` read access for the logical erase block is needed and ensured by the

concurrent component *EBM Headers*
state $apebs$: `Array`⟨`APeb`⟩
ghost state $opebs$: `Array`⟨`Owner`⟩, $oheaders$: `Owner`
ownership
$apebs[p]$.`echdr` **owned by** $oheaders$
$apebs[p]$.`vidhdr` **owned by** $oheaders$
$apebs[p]$.`bad` **owned by** $oheaders$
$apebs[p]$.`data` **owned by** $opebs[p]$
$apebs[p]$.`written` **owned by** $opebs[p]$
concurrent invariant
`ebm-io-inv`$(apebs) \wedge \# apebs = \# opebs$
rely
$\# apebs'' = \# apebs'$
$\wedge \forall p.$ `read`?$(tid, opebs'[p]) \rightarrow$ $apebs''[p]$.`bad` $= apebs'[p]$.`bad`
$\wedge\ apebs''[p]$.`echdr` $= apebs'[p]$.`echdr`
$\wedge\ apebs''[p]$.`vidhdr` $= apebs'[p]$.`vidhdr`
interface operations
`ebm_io_read_echeader`$(p; aehdr, isbflip, err)$
pre $p < \# apebs \wedge \neg\ apebs[p]$.`bad` $\wedge$ `read`?$(tid, oheaders)$
`ebm_io_write_echeader`$(p, aehdr; err)$
pre $p < \# apebs \wedge \neg\ apebs[p]$.`bad` $\wedge\ apebs[p]$.`echdr` $=$ `empty`
$\wedge$ `write`?$(tid, oheaders) \wedge$ `write`?$(tid, opebs[p])$
`ebm_io_read_data`$(p, poff, boff, len; buf, isbflip, err))$
pre $p < \# apebs \wedge \neg\ apebs[p]$.`bad` $\wedge \ldots \wedge$ `read`?$(tid, oheaders)$
`ebm_io_write_data`$(p, poff, boff, len, buf; err)$
pre $p < \# apebs \wedge \neg\ apebs[p]$.`bad` $\wedge\ apebs[p]$.`echdr.valid`? $\wedge\ apebs[p]$.`vidhdr.valid`?
$\wedge \ldots \wedge$ `write`?$(tid, oheaders) \wedge$ `write`?$(tid, opebs[p])$
$\vdots$

Figure 14.2: Concurrent Component *EBM Headers*

additional precondition `read`?$(ovols[v][l], tid)$.

The only simplification to the interface, compared to the sequential version in Fig. 10.2 on page 94, is that currently volume creation is performed in the initialization, which matches the usage in the Flashix file system. The rely condition (rely-*AEBM*) together with the canonical rely (ownership-rely) on page 181 associated with the ownership annotation (ownership-*AEBM*) make a proof of stability of the preconditions and a proof of the invariant possible (see Lem. 4.31 on page 43 for the proof obligations).

$$dom(avols'') = dom(avols') \wedge \forall v.\ \# avols''[v] = \# avols'[v] \qquad \text{(rely-}AEBM\text{)}$$

14.2 Specification of Concurrent Header Serialization

The erase block manager uses the subcomponent *EBM Headers* to write the EC- and VID-header of each erase block. The component *EBM Headers* uses ownership annotations and ghost state to profit from the synchronization provided by the erase block manager. However, the ownership model is more complicated as shown in Fig. 14.2. The general idea is that access to the EC- and VID-header and the flag `bad` is only performed by a single thread of the client component *EBM*, whereas the data pages of an erase block may be accessed by several threads concurrently.

The fields `echdr`, `vidhdr` and `bad` of each physical erase block are owned by the global owner *oheaders*, whereas all other fields of the PEB p are owned by $opebs[p]$. An access to the fields `data` and `written` for reading and writing therefore need to acquire ownership of

concurrent component *EBM*
  subcomponent *EBM Headers* (see Fig. 14.2 on page 185)
state $vols\colon \mathbb{V} \nrightarrow \texttt{Array}\langle\texttt{PebMapping}\rangle$, $ppa\colon \texttt{Array}\langle\texttt{PebProps}\rangle$,
  $eraseq\colon \texttt{List}\langle\texttt{EraseInfo}\rangle$, $sqnum\colon \mathbb{N}$, $bflips\colon \texttt{Set}\langle\mathbb{N}\rangle$, $doWl\colon \mathbb{B}$
  $volspeb\colon \mathbb{N}$, $free\colon \texttt{Set}\langle\texttt{TreeEntry}\rangle$, $used\colon \texttt{Set}\langle\texttt{TreeEntry}\rangle$
  $glock\colon \texttt{mutex}$, $vollocks\colon \mathbb{V} \nrightarrow \texttt{Array}\langle Rwl\rangle$
ghost state $ovols\colon \mathbb{V} \nrightarrow \texttt{Array}\langle\texttt{Owner}\rangle$
ownership
  $vols[v][l]$ **owned by** $vollocks[v][l].\texttt{ownership}$
  $ppa, eraseq, sqnum, bflips, doWl, volspeb, free, used$ **owned by** $glock.\texttt{ownership}$
rely
  $dom(vols'') = dom(vols') \wedge (\forall v.\ \#\, vols''[v] = \#\, vols'[v])$
  $\wedge\ (\texttt{read?}(tid', glock'.\texttt{ownership}) \rightarrow vols'' = vols')$
  $\wedge$ (LEB-ownership)
concurrent invariant
  $dom(vols) = dom(vollocks) = dom(ovols) \wedge (\forall v.\ \#\, vollocks[v] = \#\, vols[v] = \#\, ovols[v])$
  $\wedge\ \texttt{inj}(vols)$ $\wedge$ (mapped-peb-valid) $\wedge$ (ownership-transfer) $\wedge$ (seq-inv)

Figure 14.3: Concurrent Component *EBM*: State, Invariants and Rely Conditions

the single PEB p via $opebs[p]$ only. A read access to the headers and the flag `bad` requires access granted by the global owner *oheaders*. For write access to those fields of a PEB p, access granted by both *oheaders* and $opebs[p]$ are required.

The rely condition given in Fig. 14.2 and the canonical rely condition (ownership-rely) on page 181 associated with the ownership annotations of the figure again ensure that the preconditions of all operations are stable over the steps of other threads and that the invariant holds.

14.3 Concurrent Wear-Leveling

The erase block manager essentially uses two types of locks. One global mutex *glock* that must be acquired if

- one of the state variables of the component *EBM* is written, or
- one of the unmapped physical erase blocks is written, or
- one of the headers or the flag `bad` of any physical erase block is read or written.

The second type of lock is a reader/writer lock $vollocks[v][l]$ per logical erase block, which guards reads and writes to the data pages of the mapped erase block. The order of locking is that $vollocks[v][l]$ is acquired before *glock*, in the case both locks need to be acquired.

Fig. 14.3 shows the state variables, invariants and rely conditions of the component. The invariants and rely conditions ensure that for each logical erase block there always exists a corresponding lock in *vollocks*. Technically, ownership of $vols[v][l]$ is acquired by the lock $vollocks[v][l]$, however, write access is only performed when *glock* is also held. Therefore the rely states that the entire forward mapping is unmodified if the mutex *glock* is held by the thread. Note that interference is still possible, but essentially restricted to the physical erase blocks when the lock *glock* is held.

The state variable *ovols* is inherited by the specification *AEBM* of the erase block manager (Fig. 14.1 on page 184). It serves the same purpose, i.e., clients of the erase block manager first acquire read and write access to a *logical* erase block before the corresponding read and write operations may be called. Fig. 14.4 shows the implementation of each of the operations.

The figure also shows a simplified, concurrent version of the write operation. The **green** parts of the code are added. First, the reader/writer lock for the logical erase block is acquired.

interface operations

`ebm_rlock`(v, l)
  pre $\langle v, l\rangle \in ovols$
  if* $ovols[v][l]$.`readers`? **then** $ovols[v][l] \coloneqq$ `readers`$(ovols[v][l]$.`tids` $+\!\!+\ tid)$;
`ebm_wlock`(v, l)
  pre $\langle v, l\rangle \in ovols$
  if* $ovols[v][l] =$ `readers`$(\emptyset)$ **then** $ovols[v][l] \coloneqq$ `writer`(tid);
`ebm_unlock`(v, l)
  pre $\langle v, l\rangle \in ovols$
  if* $tid \in ovols[v][l]$.`readers` **then** $ovols[v][l] \coloneqq$ `unlock`$(ovols[v][l], tid)$;
`ebm_write`$(v, l, poff, boff, len, buf; err)$
  pre $\langle v, l\rangle \in vols \wedge vols[v][l]$.`mapped`? $\wedge\ poff + len \leq$ `LEB_SIZE` $\wedge\ boff + len \leq \#\ buf$
    $\wedge$ `page-aligned`$(poff) \wedge$ `page-aligned`$(len) \wedge vols[v][l]$.`written` $\leq poff$
    $\wedge\ tid \in ovols[l][v]$.`writers`
  `rwlock_wlock`$(tid; vollocks[v][l]) : \mathbb{R}$;
  let $p = vols[v][l]$.`peb` **in**
    assert $opebs[p] =$ `readers`$(\emptyset)$;
    `aebm_io_wlock`$(p) : \mathbb{R}$;
    `aebm_io_write_data`$(p, poff, boff, len, buf; err)$;
    assert $opebs[p] =$ `writer`(tid);
    `aebm_io_unlock`$(p) : \mathbb{L}$;
  `rwlock_unlock`$(tid; vollocks[v][l]) : \mathbb{L}$;

Figure 14.4: Concurrent Component *EBM*: Acquiring and Releasing Ownership and Writing to a *Logical* Erase Block

Afterwards, the mapping is read and ownership for the physical erase block is acquired. After the actual write, ownership to the physical erase block is relinquished and the reader/writer lock is released at the end. The precondition of the call to `aebm_io_write_data` is ensured by Invariant (mapped-peb-valid).

concurrent invariant (mapped-peb-valid)

$$\forall \langle v, l\rangle \in vols.\ vols[v][l] \neq \texttt{unmapped} \rightarrow \neg\, apebs[vols[v][l].\texttt{peb}].\texttt{bad} \wedge apebs[vols[v][l].\texttt{peb}].\texttt{echdr.valid?} \wedge apebs[vols[v][l].\texttt{peb}].\texttt{vidhdr.valid?}$$

As highlighted in the figure, before write may be called ownership to the logical erase block has to be acquired via `ebm_wlock`. This ensures that the rest of the precondition of write is stable over the steps of other threads, as shown next.

The rely condition associated with the ownership of a logical erase block in the component *EBM* is given by (LEB-ownership). This ensures that if a client thread has acquired read access to a logical erase block, then the mapped PEB may be changed, but whether or not a mapping exists remains unchanged. This ensures that the precondition of `ebm_map` (see Fig. 14.5 below) is stable over the steps of other threads. Additionally, the number of bytes written to the currently mapped PEB may only decrease (due to wear-leveling), but may never increase. This ensures that the precondition of `ebm_write` (see Fig. 14.4) is also stable.

rely (LEB-ownership)

$$\forall \langle v, l\rangle \in vols'.\ tid' \in \texttt{readers}(ovols'[v][l]) \rightarrow \big(vols'[v][l] = \texttt{unmapped} \leftrightarrow vols''[v][l] = \texttt{unmapped}\big) \wedge \big(vols'[v][l] \neq \texttt{unmapped} \rightarrow pebs'[vols'[v][l].\texttt{peb}].\texttt{written} \geq pebs''[vols''[v][l].\texttt{peb}].\texttt{written}\big)$$

interface operations

```
ebm_map(v, l; err)
  pre ⟨v, l⟩ ∈ vols ∧ ¬ vols[v][l].mapped? ∧ tid ∈ ovols[l][v].writers
  rwlock_wlock(tid; vollocks[v][l]) : ℝ;
  mutex_lock(tid; glock) : ℝ;
  let p = 0 in
    ebm_allocate_peb(; p, err);
    if err = ESUCCESS then
      assert opebs[p] = readers(∅);
      aebm_io_wlock(p) : ℝ;
      assert oheaders = readers(∅);
      aebm_io_header_lock() : ℝ;
      aebm_io_write_vidheader(p, mkavidhdr(v, l, sqnum, 0, 0); err);
      sqnum := sqnum + 1;
      if err = ESUCCESS then
        vols[v][l] := mapped(p), ppa[p].state := USED, used :+= tree-entry(p, ppa[p].ec);
      else
        ebm_asynchronous_erase(p, None);
      assert oheaders = writer(tid);
      aebm_io_header_unlock() : 𝕃;
      assert opebs[p] = writer(tid);
      aebm_io_unlock(p) : 𝕃;
  mutex_unlock(tid; glock) : 𝕃;
  rwlock_unlock(tid; vollocks[v][l]) : 𝕃;
```

Figure 14.5: Concurrent Component *EBM*: Mapping a Logical Erase Block

The only steps that change $vols[v][l]$ are in during mapping, unmapping and wear-leveling. During mapping and unmapping the thread that changes $vols[v][l]$ holds write ownership to $ovols[v][l]$ and no other thread can hold read access and (LEB-ownership) becomes vacuous. For wear-leveling the mapping is changed, but it is shown that the number of written bytes does not increase (see Sec. 10.5). All other writes to a PEB p do not affect the mapped PEB $vols[v][l]$.peb, because 1) the forward mapping $vols$ is always kept injective and therefore no other concurrent write operation of a client modifies $vols[v][l]$.peb, and 2) all internal writes of the component only affect unmapped PEBs.

concurrent invariant inj($vols$)

The operation ebm_read is analogous to ebm_write. The operation ebm_map is shown in Fig. 14.5. It acquires both locks in order to gain access to all the internal data structures: The PEB is allocate from the free tree *free*, its status is set USED in the PEB Property Array, it is added to the used tree *used* or to the erase queue. In order to write the VID-header, access to all headers is acquired via a call to aebm_io_header_lock and to the PEB p via aebm_io_wlock.

Figures 14.5 and 14.4 depict the additional, user-supplied assertions and mover annotation, too. The assertions hold when the execution reaches them, due to the additional Invariant (ownership-transfer). The invariant ensures that the corresponding locks are taken before access to the corresponding data pages of a PEB or the headers is requested.

concurrent invariant (ownership-transfer)

$oheaders \subseteq glock.\texttt{ownership}$

$\wedge\, \forall l \in \texttt{unmapped-pebs}(vols).\ opebs[l] \subseteq glock.\texttt{ownership}$

$\wedge\, \forall \langle v, l\rangle \in vols.\ vols[v][l] \neq \texttt{unmapped} \rightarrow opebs[vols[v][l].\texttt{peb}] \subseteq vollocks[v][l].\texttt{ownership}$

internal operations
`ebm_wear_leveling`()
guard *doWl*
let *from*, *avhdr*, *isbflip*, *err*, *isScrubbing* **in**
`mutex_lock`(*tid*; *glock*) : $\mathbb{R}$;
`ebm_get_wear_leveling_source_peb`(; *from*; *err*, *isScrubbing*);
if *err* = `ESUCCESS` **then**
assert *oheaders* = `readers`($\emptyset$);
`aebm_io_header_lock`() : $\mathbb{R}$;
`aebm_io_read_vidheader`(*from*; *avhdr*, *isbflip*, *err*);
assert *oheaders* = `writer`(*tid*);
`aebm_io_header_unlock`() : $\mathbb{L}$;
`mutex_unlock`(*tid*; *glock*) : $\mathbb{L}$;
if *err* = `ESUCCESS` **then**
`rwlock_wlock`(*tid*; *vollocks*[*avhdr*.`vol`][*avhdr*.`leb`]) : $\mathbb{R}$;
`mutex_lock`(*tid*; *glock*) : $\mathbb{R}$;
⋮
`mutex_unlock`(*tid*; *glock*) : $\mathbb{L}$;
`rwlock_unlock`(*tid*; *vollocks*[*avhdr*.`vol`][*avhdr*.`leb`]) : $\mathbb{L}$;

Figure 14.6: Concurrent Component *EBM*: Wear-Leveling

where $o_0 \subseteq o_1$ is defined by

$$\texttt{readers}(tids_0) \subseteq o_1 \leftrightarrow tids_0 = \emptyset \vee \exists tids_1.\ tids_0 \subseteq tids_1 \wedge o_1 = \texttt{readers}(tids_1)$$
$$\texttt{writer}(tid_0) \subseteq o_1 \leftrightarrow o_1 = \texttt{writer}(tid_0)$$

The assertions ensure that the annotated mover type can be proven in a separate proof obligation. For all other statements a both-mover is automatically inferred, since they only use input/output parameters, local variables or state variables with ownership expression.

The other operations of the component are extended similarly. The only difficult operation is wear-leveling as shown by Fig. 14.6. In order to read the VID-header of the source PEB, it is necessary to take the global lock. After the mapping is read, the lock must first be released. Then the lock of the mapping and the global lock are acquired, in this order to ensure deadlock-freedom. Afterwards, it is checked that the mapping still points to the PEB *from*, since an unmap operation could already have removed the PEB's mapping. Only then the actual wear-leveling is performed.

After atomicity refinement the wear-leveling operation essentially essentially consists of two atomic blocks. The first block just reads some state and the core of wear-leveling is performed in the second atomic block.

The Invariants (seq-inv) of the sequential verification of Ch. 10 still hold, however, only if the global lock *glock* is free.[1]

concurrent invariant (seq-inv)

$$\begin{aligned} glock = \texttt{free} \rightarrow \quad & (\text{ppa-inv}) \wedge (\text{unique-sqns}) \wedge (\text{vols-inv}) \wedge (\text{bflips-inv}) \wedge (\text{eraseq-no-dups}) \\ & \wedge (\text{eraseq-ppa-inv}) \wedge (\text{eraseq-mapping}) \wedge (\text{used-inv}) \wedge (\text{free-inv}) \\ & \wedge (\text{sqnum-inv}) \wedge \ (\text{vtbl-inv}) \end{aligned}$$

Note that this invariant is too weak for a direct proof of refinement for the reset transition. An atomicity refinement, however, ensures that once the critical section protected by the global

[1] Note that additional rely conditions are necessary to ensure that PEB validity and maximality of mapped PEBs is maintained by concurrent writers. It is also possible to use weaker invariants that omit the inverse mapping entirely for the rely/guarantee proofs and establish the strong invariants after the atomicity refinement with a sequential verification.

lock *glock* is atomic, the additional invariant *glock* = `free` holds and a proof of refinement of the reset transition becomes possible.

Theorem 17 (Correctness & Crash-Safety of Concurrent Wear-Leveling)**.** *The concurrent component EBM refines the concurrent component AEBM.*

Proof. A proof of the obligations of Lem. 4.31 on page 43 implies that the invariants and assertions hold, and that the component is divergence-free. Afterwards, a more atomic component *EBM'* is extracted from component *EBM* by applying the Lipton reduction of Thm. 16 on page 178 twice. In the first step the inner mutex *glock* yields atomic sections. Afterwards, *glock* = `free` is an invariant and the guards of the atomic section can be removed by Lem. 13.3 on page 179. A second application of Thm. 16 on page 178 yields the component *EBM'* where all operations, except wear-leveling, are atomic and wear-leveling essentially consists of two steps. By the theorem $EBM \sqsubseteq EBM'$ holds. The component EBM' then has the invariants that all locks $vollocks[v][l]$ are free and the invariants of the sequential component *EBM* of Ch. 10.

With the forward simulation (ebm-abs) on page 117 and the invariants *inv* of the sequential component *EBM* of Ch. 10 and the rely conditions of this chapter, we proof that $EBM' \sqsubseteq AEBM$ holds. For all operations except wear-leveling we prove the standard data refinement proof obligations of Thm. 3 on page 59. For wear-leveling we prove a commuting diagram with the proof obligation (wl-refine), where $\underline{CS}$ denotes the combined state of components *EBM* and *EBM Headers*, and $\underline{AS}$ is the state of the specification component *AEBM*.

$$
\begin{aligned}
&\Box\ \big(inv(\underline{CS}) \wedge inv(\underline{CS'})\big), \Box\, rely(Tid', \underline{CS'}, \underline{CS''}), abs(\underline{AS}, \underline{CS}), \qquad \text{(wl-refine)}\\
&\Box\, \underline{AS'} = \underline{AS}, \Box\, Tid'' = Tid'\\
&\vdash \langle abs(\underline{AS'}, \underline{CS'}) \rightarrow abs(\underline{AS''}, \underline{CS''}),\ abs(\underline{AS}, \underline{CS}) \rightarrow abs(\underline{AS'}, \underline{CS'}),\\
&\qquad true,\ false,\ \texttt{ebm\_wear\_leveling}(; \underline{CS})\rangle_{\underline{CS}, Tid}\ true
\end{aligned}
$$

The proof obligation shows that each of the steps of wear-leveling refines a **skip** step on the abstract level and propagates the abstract relation forward. □

14.4 Related Work

The flash file system by Damchoom et al. [39, 37, 36] has concurrent wear-leveling. The synchronization between threads is implicitly performed by the semantics of Event-B models, i.e., an event in an Event-B model is always executed atomically, and not explicitly via locks or other synchronization primitives. This makes the step to actual running code more difficult and less straightforward. The erase block management of this chapter is also more general, because it does not use additional bits of out-of-band data of an erase block and all data structures are stored on flash and recovered after a power failure.

15 Summary, Lessons Learned & Outlook

Summary. This chapter summarizes the contributions of this thesis and the current state of the Flashix project. Several lessons learned during the formal development of Flashix are presented and discussed. The chapter concludes with an outlook for the Flashix file system.

Contents

15.1 Summary

This thesis contributes a novel semantics for concurrent, crash-aware components with Def. 4.19 on page 34. The formalism integrates the *state-based* and *operations-based* interpretation of a power failure. The state-based view is specified by a crash predicate, whereas the operations-based view is characterized by *synchronized states* given by a synchronized predicate.

Thm. 1 on page 42 shows that the semantics is compositional and refinement is preserved by substitution under the condition of strong admissibility.

Theorem 1 (Compositionality). Given a non-retracting component C that is totally correct with respect to a specification component A, i.e., $A \sqsubseteq C$ is satisfied, and a strongly admissible component M with subcomponent A, then $M \sqsubseteq M\{A \mapsto C\}$ holds.

Compositionality is fundamental for a large-scale verification effort, because it facilitates modular and scalable reasoning. It allows for a decomposition of the complete system into smaller individual refinement steps, as shown in Fig. 6.1 on page 66 for the Flashix file system.

Three distinct types of refinement are considered and applied in this thesis: atomicity refinement, data refinement and crash refinement.

This thesis contributes a hierarchy of criteria, namely crash-recover-atomicity, crash-recover-retractability, crash-recover-introducibility and crash-recover-neutrality, ensuring that (part of) a component can be considered atomic with respect to power failures. The corresponding notion of refinement is given by *atomicity refinement* and proven correct by Thm. 2 on page 52 for sequential, crash-aware components.

Theorem 2 (Atomicity Refinement of Sequential Components). If the program p is crash-recover-atomic in the context of a component C, then $C\{p \mapsto \textbf{atomic}\ \{\, p \,\}\} \sqsubseteq C$ holds.

Atomicity refinement is extended to retraction-free, concurrent components by Thm. 16 on page 178, which combines the criteria for atomicity with respect to crashes with Lipton reductions, a criterion for atomicity with respect to concurrent threads.

Theorem 16 (Atomicity Refinement of Retraction-Free, Concurrent Components). Given a reducible program p of a non-retracting and divergence-free component C, then

$$C\{p \mapsto \textbf{atomic}\ \varphi\ \{\, p \,\}\} \sqsubseteq C$$

holds, where φ is the guard of the first statement of p if it is an atomic block or $\varphi \equiv \mathit{true}$ if the first statement is not an atomic block.

The integration of Lipton reductions and ownership annotations to sufficiently automate the approach of Thm. 16 into the tool KIV constitutes a contribution of this thesis, too. Ownership annotations are used to automatically generate assertions, which are proven to hold in a rely/guarantee proof. The assertions ensure that certain types of statements are automatically classified as both-movers. A sequence of right- and left-movers can be combined into an atomic section while maintaining liveness. The concept of movers is extended with the atomicity criteria of Ch. 5 to ensure atomicity with respect to concurrent threads as well as power failures. The increase in atomicity has the effect that stronger invariants can be established afterwards. This makes a proof of data refinement possible. Especially the proof obligation for the transition of the power failure and subsequent recovery only refines an abstract specification given very strong invariants, as shown for the example of the erase block manager in Ch. 10 and Ch. 14.

It is also proven by Thm. 3 on page 59 that standard data refinement carries over to specification components with the novel semantics. Two additional proof obligations are necessary. The first proof obligation ensures that a power failure with subsequent recovery refines the abstract specification. The second proof obligation shows that synchronized states are respected. The latter proof obligation is trivial to prove, but ensures that the operations-based interpretation of power failure propagates over an entire component hierarchy implicitly.

Theorem 3 (Data Refinement). A refinement $A \sqsubseteq C$ of two specification components A and C is implied by a forward simulation $\mathit{abs}(\underline{x}^A, \underline{x}^C)$ satisfying the conditions (1.) to (5.).

$\vdots$

4. $\mathit{sync}^A(\underline{x}^A) \wedge \mathit{abs}(\underline{x}^A, \underline{x}^C) \rightarrow \mathit{sync}^C(\underline{x}^C)$
5. $\mathit{abs}(\underline{x}_0^A, \underline{x}_0^C) \wedge \mathit{crash}^C(\underline{x}_0^C, \underline{x}_1^C)$
   $\rightarrow \langle\!|\, \mathtt{recover}^C(; \underline{x}_1^C) \,|\!\rangle \left(\exists \underline{x}_1^A.\ \mathit{crash}^A(\underline{x}_0^A, \underline{x}_1^A) \wedge \langle \mathtt{recover}^A(; \underline{x}_1^A) \rangle\, \mathit{abs}(\underline{x}_1^A, \underline{x}_1^C) \right)$

The operations-based interpretation is introduced into a component hierarchy by a novel type of refinement named *crash refinement*. Thm. 4 on page 61 gives sufficient conditions when (part of) the effect of the crash predicate can be reinterpreted in terms of synchronized states characterized by the synchronized predicate.

Theorem 4 (Crash Refinement). Given two specification components A and C that only differ in their crash and synchronized predicates. Then $A \sqsubseteq C$ is implied by the proof obligations (1.) to (3.) for all interface and internal operations $\mathtt{Op} \equiv \mathtt{Op}^C \equiv \mathtt{Op}^A$.

1. $\left(\exists \underline{x}_2.\ \mathit{crash}^A(\underline{x}_0, \underline{x}_2)\right) \wedge \mathit{crash}^C(\underline{x}_0, \underline{x}_1) \rightarrow \mathit{crash}^A(\underline{x}_0, \underline{x}_1)$
2. $\mathit{sync}^A(\underline{x}) \rightarrow \mathit{sync}^C(\underline{x})$
3. $\langle \mathtt{Op}^C(\mathit{in}; \underline{x}_0, \mathit{out}) \rangle\, \mathit{crash}^C(\underline{x}_0, \underline{x}_1)$
   $\rightarrow \mathit{crash}^C(\underline{x}_0, \underline{x}_1) \vee \left(\langle \mathtt{Op}^A(\mathit{in}; \underline{x}_0, \mathit{out}') \rangle\, \mathit{crash}^A(\underline{x}_0, \underline{x}_1) \right)$

This thesis also contributes a novel correctness criterion (Def. 7.1 on page 81) for the POSIX specification, termed *quasi-sequential crash-consistency* in the style of Bornholt et al. [18].

The theory is applied to the Flashix file system. The component hierarchy of Flashix is shown in Fig. 6.1 on page 66. For the entire file system two reductions given by Thm. 5 on page 72 and Thm. 6 on page 73 are used, which ensure that all components are crash-introducible by a simple proof obligation about the specification subcomponents. Both theorems extend previous work by the author's colleague Ernst [43] by applying the more general calculus for crash-atomicity presented in Ch. 5.

Theorem 5 (RAM Components & Atomicity). All operations of a RAM component M with subcomponent A are crash-introducible (and thereby crash-atomic) if all operations of A are crash-introducible.

Theorem 6 (Crash-Introducible Specification Components). An operation $\mathtt{Op}_i^A$ of a specification component A with state $\underline{x}$, precondition $pre(\underline{in}, \underline{x})$, invariant $inv(\underline{x})$ and crash predicate $crash(\underline{x}, \underline{x}')$ is crash-introducible if the proof obligation

$$pre(\underline{in}, \underline{x}) \wedge inv(\underline{x}) \wedge crash(\underline{x}, \underline{x}') \rightarrow \langle \mathtt{Op}_i^A(\underline{in}; \underline{x}, \underline{out}) \rangle\, crash(\underline{x}, \underline{x}')$$

holds.

With respect to the Flashix file system, several of the components (see Fig. 6.1 on page 66 for an overview over the components) and their proof of correctness and crash-safety are contributed by this thesis.

Thm. 7 on page 118 and Thm. 8 on page 120 show that the erase block manager is correct and that wear-leveling improves the distribution of erase cycles across the flash device.

Theorem 7 (Correctness & Crash-Safety of Erase Block Management & Wear-Leveling). The component *EBM* refines the component *AEBM*, and the component *Header Serialization* refines the component *EBM Headers*.

Theorem 8 (Quality of Wear-Leveling). A successful run of wear-leveling leads to a better distribution of erase counters.

Thm. 9 on page 135 shows that transactions are implemented correctly and are atomic with respect to power failures, and that garbage collection is correct. Thm. 10 on page 140 proves that garbage collection chooses a reasonable block.

Theorem 9 (Correctness & Crash-Safety of Journaling & Garbage Collection). The component *Transactions* refines the component *Index* & *Journal*.

Theorem 10 (Quality of Garbage Collection). The garbage collection algorithm of component *Transactions* chooses a logical erase block with the least amount of bytes still referenced by the RAM index.

Not only garbage collection of the journal is performed. The LEBs allocated for the index are also garbage collected by the component *Node Serialization* and Thm. 11 on page 159.

Theorem 11 (No Unused Nodes After Commit). After the commit of the file system, no unused LEBs are mapped.

The serialization of individual nodes is atomic with respect to crashes by Thm. 12 on page 160.

Theorem 12 (Correctness & Atomicity of Node Serialization). The component *Persistence* is refined by the component *Node Serialization*.

The write-buffer is also correct by Thm. 13 on page 162. Furthermore, Thm. 14 on page 163 ensures the correctness of the switch from the state-based interpretation of the crash behavior of the write-buffer to the operations-based interpretation.

Theorem 13 (Correctness & Crash-Safety of the Write-Buffer). The component *Wbuf* refines the component *Wbuf* (*State-based*).

Theorem 14 (State-Based & Operations-Based Interpretation of the Write-Buffer). The component *Wbuf* (*State-based*) is a crash refinement of the component *Wbuf* (*Op-based*).

The operations-based interpretation is then the lever to propagate the effect of the crash behavior of the write-buffer implicitly upwards the entire refinement hierarchy. This simplified the specification and verification of all components above the write-buffer significantly. For the two abstractions directly above the write-buffer (*Persistence* and *Transactions*) there is a decrease of 40% resp. 17% of user interactions in the proofs (from 500 to 300 and from 1270 to 1050). Those are the only levels of abstraction where a verification without the component semantics with implicit retractions was attempted. Note that at both levels it was still *naturally* possible to express the effect of a power failure with the state-based view. For all components above the component *Transactions* this is no longer possible, and it is reasonable to expect a much larger saving with respect to specification and verification for these components.

Thm. 15 on page 166 proves that the commit of the file system is atomic with respect to crashes and functionally correct.

Theorem 15 (Correctness & Crash-Safety of Commit). The commit is performed atomically, i.e., the component *Superblock* refines the specification *Commit*.

Finally, the theory of atomicity refinement of concurrent components is applied to the erase block manager and shows that wear-leveling and asynchronous erasure can be performed concurrently in the background by Thm. 17 on page 190.

Theorem 17 (Correctness & Crash-Safety of Concurrent Wear-Leveling). The concurrent component *EBM* refines the concurrent component *AEBM*.

15.2 Lessons Learned

Data Abstraction & Refinement One general lesson throughout the specification and verification of Flashix is that data abstraction and refinement are key ingredients for the development of large software systems with formal methods. One issue with refinement and abstraction, however, is the choice of the right "size" of steps.

On the one hand, a too small step might lead to a lot more specification that needs to be maintained. One major factor in the development of Flashix is keeping all models and proofs consistent throughout a large hierarchy of components. A similar observation is also made in [88], which classifies patches for Linux file systems across several metrics, where around half of the patches are classified as maintenance patches. An example encountered in the Flashix project is that previous iterations had an additional component called *LogFS* between component *AFS* and *FFSC* in Fig. 6.1 on page 66. The component introduced an abstract recovery mechanism based on a log into the *AFS* model. However, this did not simplify the verification significantly enough to justify the effort of keeping its interface operations and their inputs and outputs consistent with *AFS* and of maintaining the invariant and refinement proofs.

On the other hand, if the steps become too large, then the complexity in the interactive verification increases drastically, because "obvious" facts become clouded by too much noise due to the inadequate level of abstraction. One example in Flashix is the component *Header Serialization* of Ch. 10, which provides the (de-)serialization of the EC- and VID-header for the erase block manager. This component was introduced, because reasoning about subranges of the physical erase block turned out to be a lot more pervasive than initially assumed. Without this component additional subranges for the first page, the second page and the data pages are necessary in the invariants of the erase block manager. This would complicate the verification significantly, because validity of physical erase blocks as described in Sec. 10.5 in more detail depends on both headers and the data pages. This evolved to a generally beneficial pattern for Flashix where most of the (de-)serialization of the various data structures is performed in a separate component with an interface that

only refers to the algebraic data structures, sometimes with an additional value for empty or garbage data.

It is especially hard to find the right level of abstraction when additional concerns, such as power failures, play a role across component boundaries. At several points in the component hierarchy of Flashix the "natural" state of components is augmented with additional information, just to be able to express the effect of a power failure. This is prevalent in the middle of the hierarchy where the commit data structures are needed in several adjacent models in order to handle and recover from power failures. However, the theory with synchronized states and the implicit propagation of effects of the write-buffer during a power failure already significantly simplifies the specification and proofs as shown in the previous section.

In summary, the size of the steps is not only determined by the conceptual gap, but also by the project management cost and the difficulty and challenges in specification and verification. This is usually not a trade-off that is obvious or known ahead of time. Thus, the component structure of the system evolves over time, when superfluous steps are eliminated and additional steps introduced.

Maintenance & Tool Support There are several factors that lower the cost of maintenance and also simplify the initial verification. First, specification mechanisms such as components with crash and synchronized predicates, invariants, rely conditions, ownership annotations, etc. need to be first-class citizens in the tooling. This facilitates the automatic generation of proof obligations and lowers the cost of maintenance. It increases the confidence in refactorings of components as well. Another important aspect is the cost of replaying proofs. There it has proven extremely beneficial to add invariants for while-loops directly in the code for example.[1] This increases the ease of redoing a proof significantly, because it is no longer necessary to amend or modify the invariant by hand during the proof when variable names change for example.

Polymorphic Type System One way to simplify the expression of several invariants and abstraction relations presented in this thesis is a polymorphic type system. Currently, KIV has a heavy-weight, static mechanism for instantiating polymorphic types such as sets and maps. This discourages the use of higher-order operations such as `map`, `fold` and `filter`, since in general several instantiations of the polymorphic types are used in their declaration. However, in the verification of the erase block manager of Ch. 10 for example invariants expressible with these constructs are pervasive. There a truly polymorphic type system could greatly aid at least in specification. With respect to the verification, stronger simplification of higher-order lambda expressions, e.g., the predicate of a `filter` expression needs to be evaluated for a given element, might be necessary in practice.

Predicate Subtyping Another related feature is that in a large system such as Flashix usually data structures have additional constraints not expressible just by defining them as a free data type. For example physical and logical erase blocks have a fixed-size array as a field. Furthermore, they are empty above the index of the last write. In a type system with predicate subtyping such as PVS [117] it is possible to define a subtype of the original, free data type with additional constraints given by a predicate. This would save a lot of additional preconditions for lemmas, because some of them are then already implied by the type.

15.3 Outlook

Re-ordering, Write-Back Caches The semantics of components with implicit retractions for the operations-based interpretation of power failures facilitates expression of the crash behavior in terms of the history of the component for order-preserving write-back

[1] This is a feature that was previously not supported by the tool KIV.

caches. Potentially re-ordering caches can only be specified with the state-based approach. A generalization of the semantics to such caches while maintaining the compositionality of the semantics is currently being researched. The semantics of components in this thesis allow for the retry of one operation per thread only, after retraction of several operations. Two ideas are to allow a) re-execution of several operations and b) retractions of operations in between re-executed operations. This could be useful e.g. to support multiple write-buffers, one for normal operations and one for garbage collection. This could improve the performance of a concurrent garbage collector.

Concurrent Garbage Collection & Tree Traversal With respect to concurrency the next step for the Flashix file system is concurrency in the components below the transactional journal. Afterwards, garbage collection can be performed in the background by a concurrent thread. The final step is allowing concurrent calls to the POSIX interface. Here the challenge is the verification of the tree traversal of the Virtual File System Switch component.

Verified C-Code A step in a different direction is the verification of the generated C code. Here initial experiments were already performed for the sequential C code. Essentially, it is necessary to abstract C's representation of data as structures, unions and pointers to algebraic data types via a data refinement. Most likely most of the abstractions and proofs can be automatically added during the process of code generation. An interesting extension would be to reuse the ownership annotations of Ch. 13 within a C code verifier in order to prove data race freedom of the C code. VeriFast [71] as well as VCC [32] support fractional permissions and an ownership model, which ensures data race freedom.

Byte-Addressable Non-Volatile Random Access Memory An interesting and novel application of the theory developed in this thesis could be byte-addressable Non-volatile Random Access Memory (= NVRAM). This new memory technology promises to surpass the speed and longevity of flash memory and combines it with the random-access of each byte location of traditional volatile memory. Due to the more fine-grained write-access, it can be used to implement traditional highly concurrent data structures, such as Treiber's stack [128] and the Michael-Scott queue [94] in a persistent fashion. Initial results for the queue [52] and for two hash maps [122, 98] exist. There is also a (potentially inefficient) universal construction [69] that derives a persistent and linearizable data structure from a linearizable one for this kind of memory technology. There are also implementations of persistent objects with transactional semantics [31] and more recently entire file systems [34, 136, 137] specifically targeting NVRAM. An interesting question for future research is whether and how atomicity refinement of Ch. 5 and Ch. 13 is applicable to this emerging technology.

Bibliography

[1] Intel Flash File System Core Reference Guide, Version 1. Technical report, Intel Corporation, 2004.

[2] Page program addressing for MLC NAND application note, 2009. Samsung Electronics, http://www.samsung.com.

[3] The Open Group Base Specifications Issue 7, IEEE Std 1003.1, 2013 Edition, 2013. The IEEE and The Open Group, http://pubs.opengroup.org/onlinepubs/9699919799/.

[4] Open NAND Flash Interface Specification, 2014. Intel Corporation, Micron Technology Inc., Phison Electronics Corporation., Western Digital Corporation, SK Hynix Inc., Sony Corporation, http://www.onfi.org/specifications.

[5] OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.5.1, December 2017. Object Management Group, http://www.omg.org/spec/UML/2.5.1.

[6] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 2005.

[7] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[8] M.K. Aguilera and S. Frølund. Strict Linearizability and the Power of Aborting. *Technical Report HPL-2003-241*, 2003.

[9] S. Amani. *A Methodology for Trustworthy File Systems*. PhD thesis, CSE, UNSW, Sydney, Australia, 2016.

[10] S. Amani, A. Hixon, Z. Chen, C. Rizkallah, P. Chubb, L. O'Connor, J. Beeren, Y. Nagashima, J. Lim, T. Sewell, J. Tuong, G. Keller, T. Murray, G. Klein, and G. Heiserer. COGENT: Verifying High-Assurance File System Implementations. In *Proc. of Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 175–188. ACM, 2016.

[11] S. Amani and T. Murray. Specifying a Realistic File System. In *Workshop on Models for Formal Analysis of Real Systems*, pages 1–9, 2015.

[12] K.R. Apt, F. de Boer, and E.-R. Olderog. *Verification of Sequential and Concurrent Programs*. Springer, 3rd edition, 2009.

[13] G. Back. DataScript - A Specification and Scripting Language for Binary Data. In *International Conference on Generative Programming and Component Engineering*, pages 66–77. Springer, 2002.

[14] R.J. Back. A Method for Refining Atomicity in Parallel Algorithms. In *International Conference on Parallel Architectures and Languages Europe*, pages 199–216. Springer, 1989.

[15] M. Ben-Ari. The Bug That Destroyed a Rocket. *ACM SIGCSE Bulletin*, 33(2):58–59, 2001.

[16] R. Berryhill, W. Golab, and M. Tripunitara. Robust Shared Objects for Non-Volatile Main Memory. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 46. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[17] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Springer Science & Business Media, 2013.

[18] J. Bornholt, A. Kaufmann, J. Li, A. Krishnamurthy, E. Torlak, and X. Wang. Specifying and Checking File System Crash-Consistency Models. In *Proc. of Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 83–98. ACM, 2016.

[19] J. Boyland. Checking Interference with Fractional Permissions. In *International Static Analysis Symposium*, pages 55–72. Springer, 2003.

[20] M. Butler and D. Yadav. An incremental development of the Mondex system in Event-B. *Formal Aspects of Computing*, 20(1):61–77, 2008.

[21] A. Butterfield, L. Freitas, and J. Woodcock. Mechanising a formal model of flash memory. *Sci. Comput. Program.*, 74(4):219–237, February 2009.

[22] A. Butterfield and J. Woodcock. Formalising Flash Memory: First Steps. *IEEE Int. Conf. on Engineering of Complex Computer Systems*, 0:251–260, 2007.

[23] A. Butterfield and A. Ó Catháin. Concurrent Models of Flash Memory Device Behaviour. In *Formal Methods: Foundations and Applications*, volume 5902 of *Lecture Notes in Computer Science*, pages 70–83. Springer Berlin Heidelberg, 2009.

[24] X. Cai and S. Shao. An Optimization Algorithm for UBIFS Wear-Leveling. In *Intelligent Systems and Applications (ISA), 2010 2nd International Workshop on*, pages 1–4. IEEE, 2010.

[25] A. Cau and B. Moszkowski. ITL–Interval Temporal Logic, 2018. http://antonio-cau.co.uk/ITL/ (Accessed 13 September 2018).

[26] H. Chen. *Certifying a Crash-Safe File System*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, United States, 2016.

[27] H. Chen, D. Ziegler, A. Chlipala, M.F. Kaashoek, E. Kohler, and N. Zeldovich. Specifying Crash Safety for Storage Systems. In *Proc. of the Workshop on Hot Topics in Operating Systems (HotOS)*. USENIX Association, 2015.

[28] H. Chen, D. Ziegler, A. Chlipala, N. Zeldovich, and M.F. Kaashoek. Using Crash Hoare Logic for Certifying the FSCQ File System. In *Proc. of the Symposium on Operating Systems Principles (SOSP)*. ACM, 2015.

[29] R. Chen, M. Clochard, and C. Marché. A Formally Proved, Complete Algorithm for Path Resolution with Symbolic Links. *Journal of Formalized Reasoning*, 10(1), November 2017.

[30] T.-S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, and H.-J. Song. A survey of Flash Translation Layer. *J. Syst. Archit.*, 55(5-6):332–343, May 2009.

[31] J. Coburn, A.M. Caulfield, A. Akel, L.M. Grupp, R.K. Gupta, R. Jhala, and S. Swanson. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proc. of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 105–118. ACM, 2011.

[32] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A Practical System for Verifying Concurrent C. In *International Conference on Theorem Proving in Higher Order Logics*, pages 23–42. Springer, 2009.

[33] D. Comer. The Ubiquitous B-tree. *ACM Computing Surveys (CSUR)*, 11(2):121–137, 1979.

[34] J. Condit, E.B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better I/O Through Byte-Addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 133–146. ACM, 2009.

[35] R. Cox, M.F. Kaashoek, and R. Morris. Xv6, A Simple Unix-like Teaching Operating System, 2011.

[36] K. Damchoom. *An Incremental Refinement Approach to a Development of a Flash-Based File System in Event-B*. PhD thesis, University of Southampton, 2010.

[37] K. Damchoom and M. Butler. Applying Event and Machine Decomposition to a Flash-Based Filestore in Event-B. In *Proc. of the Brazilian Symposium on Formal Methods (SBMF)*, volume 5902 of *LNCS*, pages 134–152. Springer, 2009.

[38] K. Damchoom and M. Butler. Applying Event and Machine Decomposition to a Flash-Based Filestore in Event-B. In M.V. Oliveira and J. Woodcock, editors, *Formal Methods: Foundations and Applications*, pages 134–152. Springer, 2009.

[39] K. Damchoom, M. Butler, and J.-R. Abrial. Modelling and Proof of a Tree-Structured File System in Event-B and Rodin. In *Proc. of the International Conference on Formal Engineering Methods (ICFEM)*, volume 5256 of *LNCS*, pages 25–44. Springer, 2008.

[40] W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge University Press, 1998.

[41] J. Derrick and E. Boiten. *Refinement in Z and Object-Z*. Springer, 2001.

[42] T. Elmas, S. Qadeer, and S. Tasiran. A Calculus of Atomic Actions. In *ACM SIGPLAN Notices*, volume 44, pages 2–15. ACM, 2009.

[43] G. Ernst. *A Verified POSIX-Compliant Flash File System-Modular Verification Technology & Crash Tolerance*. PhD thesis, Augsburg, Universität Augsburg, 2017.

[44] G. Ernst, J. Pfähler, G. Schellhorn, D. Haneberg, and W. Reif. KIV - Overview and VerifyThis Competition. *Software Tools for Techn. Transfer*, 17(6):677–694, 2015.

[45] G. Ernst, J. Pfähler, G. Schellhorn, and W. Reif. Modular Refinement for Submachines of ASMs. In *Proc. of Alloy, ASM, B, TLA, VDM, and Z (ABZ)*, pages 188–203. Springer, 2014.

[46] G. Ernst, J. Pfähler, G. Schellhorn, and W. Reif. Inside a Verified Flash File System: Transactions & Garbage Collection. In *Proc. of Verified Software: Theories, Tools, Experiments (VSTTE)*, volume 9593 of *LNCS*, pages 73–93. Springer, 2015.

[47] G. Ernst, J. Pfähler, G. Schellhorn, and W. Reif. Modular, Crash-Safe Refinement for ASMs with Submachines. *Science of Computer Programming (SCP)*, 2016.

[48] G. Ernst, G. Schellhorn, D. Haneberg, J. Pfähler, and W. Reif. A Formal Model of a Virtual Filesystem Switch. In *Proc. of Software and Systems Modeling (SSV)*, EPTCS, pages 33–45, 2012.

[49] G. Ernst, G. Schellhorn, D. Haneberg, J. Pfähler, and W. Reif. Verification of a Virtual Filesystem Switch. In *Proc. of Verified Software: Theories, Tools, Experiments (VSTTE)*, volume 8164 of *LNCS*, pages 242–261. Springer, 2013.

[50] M. A. Ferreira, S. S. Silva, and J. N. Oliveira. Verifying Intel Flash File System Core Specification. In *Modelling and Analysis in VDM: Proceedings of the Fourth VDM/Overture Workshop*, pages 54–71, 2008.

[51] L. Freitas, J. Woodcock, and A. Butterfield. POSIX and the Verification Grand Challenge: a roadmap. In *Engineering of Complex Computer Systems, 2008. ICECCS 2008. 13th IEEE International Conference on*, pages 153–162. IEEE, 2008.

[52] M. Friedman, M. Herlihy, V. Marathe, and E. Petrank. Brief Announcement: A Persistent Lock-Free Queue for Non-Volatile Memory. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 91. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2017.

[53] E. Gal and S. Toledo. Algorithms and Data Structures for Flash Memories. *ACM computing surveys*, pages 138–163, 2005.

[54] A. Galloway, G. Lüttgen, J.T. Mühlberg, and R.I. Siminiceanu. Model-Checking the Linux Virtual File System. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 74–88. Springer, 2009.

[55] T. Gleixner, F. Haverkamp, and A. Bityutskiy. UBI—Unsorted Block Images. 2006.

[56] L.M. Grupp, A.M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P.H. Siegel, and J.K. Wolf. Characterizing Flash Memory: Anomalies, Observations, and Applications. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 24–33. IEEE, 2009.

[57] R. Guerraoui and R.R. Levy. Robust Emulations of Shared Memory in a Crash-Recovery Model. In *Distributed Computing Systems, 2004. Proceedings. 24th International Conference on*, pages 400–407. IEEE, 2004.

[58] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.

[59] F. Havasi. An Improved B+ Tree for Flash File Systems. In *SOFSEM*, pages 297–307. Springer, 2011.

[60] J. He, C.A.R. Hoare, and J. W. Sanders. Data refinement refined. In *Proc. of the European Symposium on Programming (ESOP)*, pages 187–196. Springer, 1986.

[61] M. Heisel. Specification of the Unix File System: A Comparative Case Study. *Algebraic Methodology and Software Technology*, pages 475–488, 1995.

[62] M.P. Herlihy and J.M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[63] W. H. Hesselink and M. I. Lali. Formalizing a Hierarchical File System. *Formal Aspects of Computing*, 24(1):27–44, 2012.

[64] C.A.R. Hoare. Proof of Correctness of Data Representations. *Acta Informatica*, 1(4):271–281, 1972.

[65] C.A.R. Hoare. The Verifying Compiler: A Grand Challenge for Computing Research. In *International Conference on Compiler Construction*, pages 262–272. Springer, 2003.

[66] G. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.

[67] A. Hunter. A Brief Introduction to the Design of UBIFS. 2008.

[68] INCITS. ATA/ATAPI Command Set - 2 (ACS-2), Revision 2, August 3, 2009.

[69] J. Izraelevitz, H. Mendes, and M. L. Scott. Linearizability of Persistent Memory Objects under a Full-System-Crash Failure Model. In *International Symposium on Distributed Computing*, pages 313–327. 2016.

[70] B. Jacobs, K.R.M. Leino, F. Piessens, and W. Schulte. Safe Concurrency for Aggregate Objects with Invariants. In *Software Engineering and Formal Methods (SEFM) 2005*, pages 137–146. IEEE, 2005.

[71] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. *NASA Formal Methods*, 6617:41–55, 2011.

[72] J.-M. Jazequel and B. Meyer. Design by Contract: The Lessons of Ariane. *Computer*, 30(1):129–130, 1997.

[73] C.B. Jones. *Development methods for computer programs including a notion of interference*. Oxford University Computing Laboratory, 1981.

[74] R. Joshi and G.J. Holzmann. A Mini Challenge: Build a Verifiable Filesystem. *Formal Aspects of Computing*, 19(2):269–272, 2007.

[75] E. Kang and D. Jackson. Formal Modeling and Analysis of a Flash Filesystem in Alloy. In *Proc. of Abstract State Machines, B, and Z (ABZ)*, volume 5238 of *LNCS*, pages 294–308. Springer, 2008.

[76] E. Kang and D. Jackson. Designing and Analyzing a Flash File System with Alloy. *Int. J. Software and Informatics*, 3(2-3):129–148, 2009.

[77] G. Keller, T. Murray, S. Amani, L. O'Connor, Z. Chen, L. Ryzhyk, G. Klein, and G. Heiser. File Systems Deserve Verification Too! *ACM Operating Systems Review*, 48(1):58–64, 2014.

[78] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009.

[79] F. Knight. TRIM - DRAT/RZAT clarifications for ATA8-ACS2, Revision 2, February 23, 2010.

[80] E. Koskinen and J. Yang. Reducing Crash Recoverability to Reachability. In *Proc. of Principles of Programming Languages (POPL)*, pages 97–108. ACM, 2016.

[81] R. Kumar, M.O. Myreen, M. Norrish, and S. Owens. CakeML: A Verified Implementation of ML. In *ACM SIGPLAN Notices*, volume 49, pages 179–191. ACM, 2014.

[82] X. Leroy. A Formally Verified Compiler Back-end. *Journal of Automated Reasoning*, 43(4):363, 2009.

[83] P. Levart. Java bindings for FUSE. http://sourceforge.net/projects/fuse-j/.

[84] N.G. Leveson and C.S. Turner. An Investigation of the Therac-25 Accidents. *Computer*, 26(7):18–41, 1993.

[85] P. Liggesmeyer. *Software-Qualität: Testen, Analysieren und Verifizieren von Software*. Springer Science & Business Media, 2009.

[86] J.L. Lions. ARIANE 5 Flight 501 Failure, Report by the Inquiry Board, July 1996. Paris.

[87] R. J. Lipton. Reduction: A Method of Proving Properties of Parallel Programs. *Communications of the ACM*, 18(12):717–721, 1975.

[88] L. Lu, A.C. Arpaci-Dusseau, R.H. Arpaci-Dusseau, and S. Lu. A Study of Linux File System Evolution. *ACM Transactions on Storage (TOS)*, 10(1):3, 2014.

[89] D. Ma, J. Feng, and G. Li. A Survey of Address Translation Technologies for Flash Memories. *ACM Computing Surveys (CSUR)*, 46(3):36, 2014.

[90] O. Marić and C. Sprenger. Verification of a Transactional Memory Manager under Hardware Failures and Restarts. In *Proc. of Formal Methods (FM)*, volume 8442 of *LNCS*, pages 449–464. Springer, 2014.

[91] P.J. McCann and S. Chandra. Packet Types: Abstract Specification of Network Protocol Messages. *ACM SIGCOMM Computer Communication Review*, 30(4):321–333, 2000.

[92] A.A. McEwan and J. Woodcock. Unifying Theories of Interrupts. In *Unifying Theories of Programming*, pages 122–141, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[93] B. Meyer. Applying 'design by contract'. *Computer*, 25(10):40–51, 1992.

[94] M.M. Michael and M.L. Scott. Simple, Fast, and Practical Non-blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of Distributed Computing*, pages 267–275. ACM, 1996.

[95] K.H. Möller. Ausgangsdaten für Qualitätsmetriken—Eine Fundgrube für Analysen. In *Software-Metriken in der Praxis*, pages 105–116. Springer, 1996.

[96] C. Morgan and B. Sufrin. Specification of the UNIX filing system. *IEEE Transactions on Software Engineering*, (2):128–142, 1984.

[97] J.T. Mühlberg and G. Lüttgen. Verifying Compiled File System Code. *Formal Aspects of Computing*, 24(3):375–391, 2012.

[98] F. Nawab, J. Izraelevitz, T. Kelly, C.B. Morrey III, D. R. Chakrabarti, and M.L. Scott. Dalí: A Periodically Persistent Hash Map. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 91. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2017.

[99] T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer, 2002.

[100] W.D. Norcott and D. Capps. Iozone filesystem benchmark, 2003.

[101] G. Ntzik, P. da Rocha Pinto, and P. Gardner. Fault-Tolerant Resource Reasoning. In *Proc. of the Asian Symposium on Programming Languages and Systems (APLAS)*, volume 9458 of *LNCS*, pages 169–188. Springer, 2015.

[102] L. O'Connor-Davis, G. Keller, S. Amani, T. Murray, G. Klein, Z. Chen, and C. Rizkallah. CDSL Version 1: Simplifying Verification with Linear Types. Technical report, NICTA, Sydney, Australia, 2014.

[103] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala: A Comprehensive Step-by-Step Guide*. Artima Press, third edition, 2016.

[104] S. Owicki and D. Gries. An Axiomatic Proof Technique for Parallel Programs. *Acta Informatica*, 6(4):319–340, 1976.

[105] S. Owicki and D. Gries. Verifying Properties of Parallel Programs: An Axiomatic Approach. *Communications of the ACM*, 19(5):279–285, 1976.

[106] C.H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, 1979.

[107] J. Pfähler, G. Ernst, S. Bodenmüller, G. Schellhorn, and W. Reif. Modular Verification of Order-Preserving Write-Back Caches. In *IFM: 13th International Conference, 2017, Proceedings*, pages 375–390. Springer, 2017.

[108] J. Pfähler, G. Ernst, G. Schellhorn, D. Haneberg, and W. Reif. Formal specification of an Erase Block Management Layer for Flash Memory. In *Haifa Verification Conference (HVC)*, volume 8244 of *LNCS*, pages 214–229. Springer, 2013.

[109] J. Pfähler, G. Ernst, G. Schellhorn, D. Haneberg, and W. Reif. Crash-Safe Refinement for a Verified Flash File System. Technical Report 2014-02, University of Augsburg, Germany, 2014.

[110] T.S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A.C. Arpaci-Dusseau, and R.H. Arpaci-Dusseau. Crash Consistency. *Queue*, 13(7):20, 2015.

[111] V. Prabhakaran, L.N. Bairavasundaram, N. Agrawal, H.S. Gunawi, A.C. Arpaci-Dusseau, and R.H. Arpaci-Dusseau. *IRON file systems*, volume 39. ACM, 2005.

[112] A. Rajgarhia and A. Gehani. Performance and Extension of User Space File Systems. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 206–213. ACM, 2010.

[113] G. Reeves and T. Neilson. The Mars Rover Spirit FLASH Anomaly. In *Aerospace Conference, 2005 IEEE*, pages 4186–4199. IEEE, 2005.

[114] J.C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55–74. IEEE, 2002.

[115] T. Ridge, D. Sheets, T. Tuerk, A. Giugliano, A. Madhavapeddy, and P. Sewell. SibylFS: Formal Specification and Oracle-Based Testing for POSIX and Real-World File Systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 38–53. ACM, 2015.

[116] M. Rosenblum and J.K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[117] J. Rushby, S. Owre, and N. Shankar. Subtypes for Specifications: Predicate Subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, 1998.

[118] G. Schellhorn, G. Ernst, J. Pfähler, D. Haneberg, and W. Reif. Development of a Verified Flash File System. In *Proc. of Alloy, ASM, B, TLA, VDM, and Z (ABZ)*, volume 8477 of *LNCS*, pages 9–24. Springer, 2014. Invited Paper.

[119] G. Schellhorn, H. Grandy, D. Haneberg, and W. Reif. The Mondex Challenge: Machine Checked Proofs for an Electronic Purse. In *International Symposium on Formal Methods*, pages 16–31. Springer, 2006.

[120] G. Schellhorn, B. Tofan, G. Ernst, J. Pfähler, and W. Reif. RGITL: A temporal logic framework for compositional reasoning about interleaved programs. *Annals of Mathematics and Artificial Intelligence*, 71(1):131–174, 2014.

[121] A. Schierl, G. Schellhorn, D. Haneberg, and W. Reif. Abstract Specification of the UBIFS File System for Flash Memory. In *Proc. of Formal Methods (FM)*, volume 5850 of *LNCS*, pages 190–206. Springer, 2009.

[122] D. Schwalb, M. Dreseler, M. Uflacker, and H. Plattner. NVC-Hashmap: A Persistent and Concurrent Hashmap For Non-Volatile Memories. In *Proceedings of the 3rd VLDB Workshop on In-Memory Data Management and Analytics*. ACM, 2015.

[123] H. Sigurbjarnarson, J. Bornholt, E. Torlak, and X. Wang. Push-Button Verification of File Systems via Crash Refinement. In *Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2016.

[124] M. Sivathanu, A.C. Arpaci-Dusseau, R.H. Arpaci-Dusseau, and S. Jha. A Logic of File Systems. In *FAST*, volume 5, 2005.

[125] M. Szeredi. File System in User Space. http://github.com/libfuse/libfuse.

[126] Y.K. Tan, M.O. Myreen, R. Kumar, A. Fox, S. Owens, and M. Norrish. A New Verified Compiler Backend for CakeML. In *ACM SIGPLAN Notices*, volume 51, pages 60–73. ACM, 2016.

[127] B. Tofan. *Compositional Concurrent Program Verification with RGITL*. PhD thesis, Augsburg, Universität Augsburg, 2014.

[128] R.K. Treiber. *Systems Programming: Coping with Parallelism*. International Business Machines Incorporated, Thomas J. Watson Research Center, 1986.

[129] H-W. Tseng, L. Grupp, and S. Swanson. Understanding the Impact of Power Loss on Flash Memory. In *Proc. of the Design Automation Conference (DAC)*, pages 35–40. ACM, 2011.

[130] S.C. Tweedie et al. Journaling the Linux ext2fs filesystem. In *The Fourth Annual Linux Expo*, 1998.

[131] UBI - Out-of-Band Data Area. http://www.linux-mtd.infradead.org/faq/ubi.html.

[132] UBIFS - Unstable Bits Issue. http://www.linux-mtd.infradead.org/doc/ubifs.html.

[133] J. Woodcock and J. Davies. *Using Z: Specification, Proof and Refinement*. Prentice Hall, 1996.

[134] J. Woodcock, S. Stepney, D. Cooper, J. Clark, and J. Jacob. The certification of the Mondex electronic purse to ITSEC Level E6. *Formal Aspects of Computing*, 20(1):5–19, 2008.

[135] D. Woodhouse. JFFS: The Journalling Flash File System. In *Ottawa Linux Symposium*, volume 2001, 2001.

[136] J. Xu and S. Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *14th USENIX Conference on File and Storage Technologies (FAST)*, pages 323–338, 2016.

[137] J. Xu, L. Zhang, A. Memaripour, A. Gangadharaiah, A. Borase, T.B. Da Silva, S. Swanson, and A. Rudoff. NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 478–496. ACM, 2017.

[138] Q. Xu, W.-P. de Roever, and J. He. The Rely-Guarantee Method for Verifying Shared Variable Concurrent Programs. *Formal Aspects of Computing*, 9(2):149–174, Mar 1997.

[139] J. Yang, N. Plasson, G. Gillis, N. Talagala, and S. Sundararaman. Don't Stack Your Log On My Log. In *INFLOW*, 2014.

[140] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using Model Checking to Find Serious File System Errors. *ACM Transactions on Computer Systems (TOCS)*, 24(4):393–423, 2006.